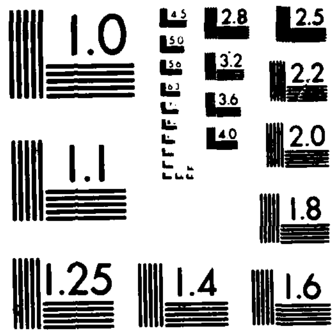
AD-A182 885 PRELIMINARY VERSION: ADA (TRADE NAME)/SQL: A STANDARD 1/3
PORTABLE ADA-DBMS I (U) INSTITUTE FOR DEFENSE ANALYSES
ALEXANDRIA VA B R BRYKCZYNSKI ET AL APR 87 IDA-P-1944
UNCLASSIFIED IDA/HQ-86-31239 MDA903-84-C-0031 F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

Copy 60 of 150 copies

AD-A182 885

DTIC FILE COPY

IDA PAPER P-1944

PRELIMINARY VERSION: Ada*/SQL: A STANDARD, PORTABLE Ada-DBMS INTERFACE

Bill R. Brykczynski
Fred Friedman

July 1986

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

IDA Log No. HQ 86-31239

87

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) P-1944	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION WIS Joint Program Office 8b OFFICE SYMBOL (if applicable) WIS JPMO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) WIS JPMO/ADT The Pentagon Washington, D.C. 20330-6600	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-W5-260 / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
Preliminary Version: Ada/SQL: A Standard, Portable Ada-DBMS Interface (U)

12 PERSONAL AUTHOR(S)
Bill R. Brykczynski, Fred Friedman

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM ______ TO ______	1987 April	238

16 SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Ada Programming Language; Structured Query Language (SQL); Data Manipulation Language (DML); Commercial Off the Shelf (COTS); Database Management System (DBMS); Tools and Techniques (Software); Binding Specifications; Portability; Reusabiliy; WWMCCS Information System (WIS; Data Definition Language (DDL).

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This IDA Paper describes the standards and software developed that have addressed the area of interfacing with the database management systems (DBMS). A standard DBMS interface has been developed, consisting of both a data definition language and a data manipulation language (DML). Use of this standard within application programs will permit them to operate transportably with any of a variety of commerical off the shelf (COTS) DBMSs. In addition, tools have been developed to assist in the implementation of the standard with a new COTS DBMS, and to automatically generate test data for use during the system checkout and tuning phases.

Ada/SQL is a binding of the proposed ANSI standard database language SQL to the Ada programming language. This binding is currently being proposed as both an ANSI and ISO standard. Ada/SQL adheres to the current version of the proposed ANSI standard for SQL as much as possible. The underlying DBMS need not, however, conform to the SQL standard; the Ada/SQL environment translates between the standard Ada/SQL interface and that of the underlying DBMS.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ■ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

IDA PAPER P-1944

PRELIMINARY VERSION: Ada/SQL: A STANDARD, PORTABLE Ada-DBMS INTERFACE

Bill R. Brykczynski
Fred Friedman

July 1986

IDA
INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-W5-206

Table of Contents

Preface

The purpose of IDA Paper P-1944, "Ada/SQL: A Standard, Portable Ada-DBMS Interface," is to communicate the results of intensive analyses into the possibility of constructing a native syntax binding specification for the Ada programming language to the database language SQL. This specification is viewed as a preliminary specification, as a complete, in-depth technical review has not, at this point, been performed. Efforts are currently underway to provide for such a technical forum, with the result being a final binding specification.

The importance of this document is based on fulfilling the objective of Task Order T-W5-206, WIS Application Software Study, which is P-1944 will be used to aid the WIS program in specifying an Ada DBMS interface. As a Paper, P-1944 is directed towards DOD, Industry and American/International standards organizations to aid in interfacing Ada to DBMS's.

This document has be reviewed by a great many individuals, too numerous to list here. However, several principal reviewers do deserve particular recognition: John Baur, LTC Terry Courtwright, Kathleen Gilroy, Dr. Michael Mendiville, Tim Porter, Dr. John Salasin, Dr. Eugen Vasilescu, and Col. Bill Whitaker.

1. Introduction

The United States Department of Defense initiative towards programming language standardization with Ada [ADA 83] shows great promise. High-level planners realize, however, that standards must go beyond the basic language definition in order to reduce the cost and time required for software development and maintenance. This report describes the standards and software developed that have addressed the area of interfacing with database management systems (DBMS). A standard DBMS interface for Ada has been developed, consisting of both a data definition language (DDL) and a data manipulation language (DML). Use of this standard within application programs will permit them to operate transportably with any of a variety of commercial, off the shelf (COTS) DBMSs. In addition, tools have been developed to assist in the implementation of the standard with a new COTS DBMS, and to automatically generate test data for use during system checkout and tuning phases.

Ada/SQL is a binding of the proposed ANSI standard database language SQL [ANSI 85] to the Ada programming language. This binding is currently being proposed as both an ANSI and ISO standard. Ada/SQL adheres to the current version of the proposed ANSI standard for SQL as much as possible. The underlying DBMS need not, however, conform to the SQL standard; the Ada/SQL environment translates between the standard Ada/SQL interface and that of the underlying DBMS.

Section 2.0 provides a comprehensive description of Ada/SQL, including discussions on portability aspects of Ada/SQL, examples of the standard and generated DDL, and examples of the standard DML. Section 3.0 provides insight on how Ada/SQL was implemented. Section 4.0 provides a list of references used in compiling this report. Appendix I describes a proposed binding of Ada to SQL (i.e. a more formal definition of Ada/SQL). Finally, Appendix I provides listings of software developed to implement a prototype version of Ada/SQL.

Note: The binding specification contained in Appendix I reflects a preliminary effort in defining Ada/SQL. The specification was developed prior to the final specification of SQL in October, 1986. Appendix III contains an addendum, with clarifications and modifications to the Ada/SQL specification, as resulted from an informal "Ada/SQL Working Group." There are currently plans to reform a similar group, with the intention of identifying potential areas where further definition or refinement is necessary. The authors would welcome any comments/suggestions that would aid in further clarification/refinement of this binding specification.

2. Description Of Ada/SQL

This section of the report provides a complete description of Ada/SQL. First, a brief introduction to the traditional components of database management systems is given. In addition, an overview of Ada/SQL and its application and tool portability concerns is presented. Next, issues pertaining to the Ada/SQL data definition language (DDL) are discussed. These issues include automatic test data generation, portability features, and examples of the data definition language. The Ada/SQL data manipulation language (DML) is presented next. A brief look at how the DML and DDL are tied together is given, followed by an overview of the implementation of the DML. An in depth discussion on portability and reusability aspects of the Ada/SQL implementation is presented next. Finally, a summary of the Ada/SQL effort ties together the goals and benefits of the Ada/SQL interface.

2.1. Introduction

This section of the report provides an introduction to issues relevant for a discussion of Ada/SQL. First, an overview of traditional DBMS components is given. Next, the implementation of these components in Ada/SQL is presented. Finally, application and tool portability issues within Ada/SQL is discussed.

2.2. Database Management System Components

Every database management system provides two main language components, one for data definition and one for data manipulation. The data definition language has traditionally been used for the one-time function of creating new databases, and perhaps for modifying existing databases where such actions were permitted. With its definitions of tables and columns (in the relational data model), it is used not only to define the structure of the database, but also to maintain the consistency of that structure and its contents. It seems reasonable to also import the data definitions into programs accessing the database, to maintain consistency between their program variable types and those within the database. This has, however, been implemented by very few relational DBMSs, undoubtedly because the languages to which they provide interfaces are not strongly typed.

Virtually all relational database management systems provide two flavors of data manipulation language, one available for program use of data and one for interactive use. It is desirable, and most DBMSs have, in fact, implemented this, that the programming language interface should be as close to the interactive interface as possible. Programmers will very often desire to use the interactive interface to experiment with various data manipulation commands during the design phases, and to set/modify data values for testing programs during later development phases. The selection of SQL as the standard DML maximizes the number of existing DBMSs with which this desired similarity between the programming language and interactive interfaces can be achieved.

2.3. Overview of the Ada/SQL DBMS Interface

The Ada/SQL interface provides these two major components, a DDL and a DML, required of any database management system.

The DDL is not, of course, used directly to define the contents of databases, since it is not the DDL of any existing database management system. It is, however, designed to be translatable into the DDLs required by typical COTS DBMSs. Any database schema written using Ada/SQL DDL will be automatically transportable to any DBMS, providing a DDL generator is available for the target DBMS. Several such DDL generators have already been written, and tools have been created that streamline their development.

The second goal for a DDL mentioned above, that of using it to define program data types consistent with database data types, is achieved by having the Ada/SQL DDL be standard Ada, compiled by any validated Ada compiler. Application programs can therefore simply "**with**" a DDL package in order to immediately and consistently have all database data types defined for them.

The DDL is also used for a third purpose, that of automatically generating test data for populating databases during the

program checkout phase. Automatically generated test data, in large quantities, is also useful for determining the performance of new DBMSs, schemas, and/or programs. Certain constructs have been built into the DDL to enable it to be used for this purpose, particularly to ensure that the test data generated is meaningful in terms of the application.

The Ada/SQL data manipulation language is, as noted above, SQL, or as close to it as is possible within the constraints of Ada syntax. Since the DML is also Ada, it may be used directly within programs that are compiled by any validated Ada compiler. In order to make this possible, the functions are defined to build data structures that are then used to translate the SQL operations into the commands required by the underlying COTS DBMS actually storing the database. A mini-DBMS that uses these data structures has been implemented in order to show how they may be used to process the SQL functions of SELECT, UPDATE, INSERT, and DELETE.

2.4. Application and Tool Portability Concerns

Ada/SQL is more than just an interface specification; it includes a set of tools for implementing the interface with an underlying DBMS. The way all these components fit together is shown in Figure 1. As can be seen, application DDL and programs are totally transportable across underlying database management systems. This portability is created by tools that translate between the Ada/SQL standard protocols and those required by a specific underlying DBMS. These tools must obviously have some components that are specific to the underlying DBMS, but they are designed such that much of their code is also transportable across DBMSs.

The DDL for a database application is written as one or more Ada packages. Application programs may "**with**" these packages to define the data types they will need to access the database. A DDL generator program reads the text of the Ada/SQL DDL to generate the DDL required to define the application database to the underlying DBMS.

Once the database has been defined, the application programs may use Ada/SQL statements to process the data stored therein. These statements are actually Ada subprograms which build data structures descriptive of the operation performed, and/or cause execution of the operation. Procedures executing Ada/SQL operations can be viewed as part of a DML converter package, which converts the Ada/SQL operations into the instructions required by the specific underlying DBMS, thereby causing the operations to actually be performed. Parts of the DML converter are transportable; the bulk of it is, however, dependent on the underlying DBMS.

The Ada/SQL data manipulation language includes references to table and column names defined by the data definition language. A SQL function generator reads the DDL and defines the necessary overloaded functions to implement these name references. Data types for strong typing of database operations are also automatically defined. The output of the SQL function generator, which consists largely of instantiations of generic functions, is "with'ed" into application programs to make the functions and data types defined visible. The data types and table/column names are independent of the underlying DBMS, so the SQL function generator is totally transportable.

The Ada/SQL DDL package may also be read as text by a test data generator tool. The test data generator uses the augmented database descriptions of the DDL to generate meaningful test data for the application programs. Since the test data generator uses Ada/SQL statements to load the database, it is totally transportable. Output can also be targeted for bulk load of a database, if warranted by the data volumes and processing speed. As already noted, large volumes of test data can also be used to derive performance figures for new DBMSs, schemas, and/or programs.

The DDL generator, SQL function generator, and test data generator all read the Ada/SQL DDL. Code to read the DDL and build descriptive data structures can be shared by all three components. The prototype was in fact implemented in this fashion, where code written for the SQL function generator was reused to write DDL generators for two different underlying DBMSs. The test data generator has not yet been prototyped.

2.5. The Ada/SQL Data Definition Language

This section of the report provides a comprehensive description of the Ada/SQL data definition language (DDL). First, the requirements of a DDL are given. Next, the concerns for automatic test data generation in Ada/SQL are enumerated. Portability issues, as well as examples of the Ada/SQL DDL, are then discussed. Finally, examples of DBMS-specific DDL, generated from the Ada/SQL Ada/SQL DDL, is presented.

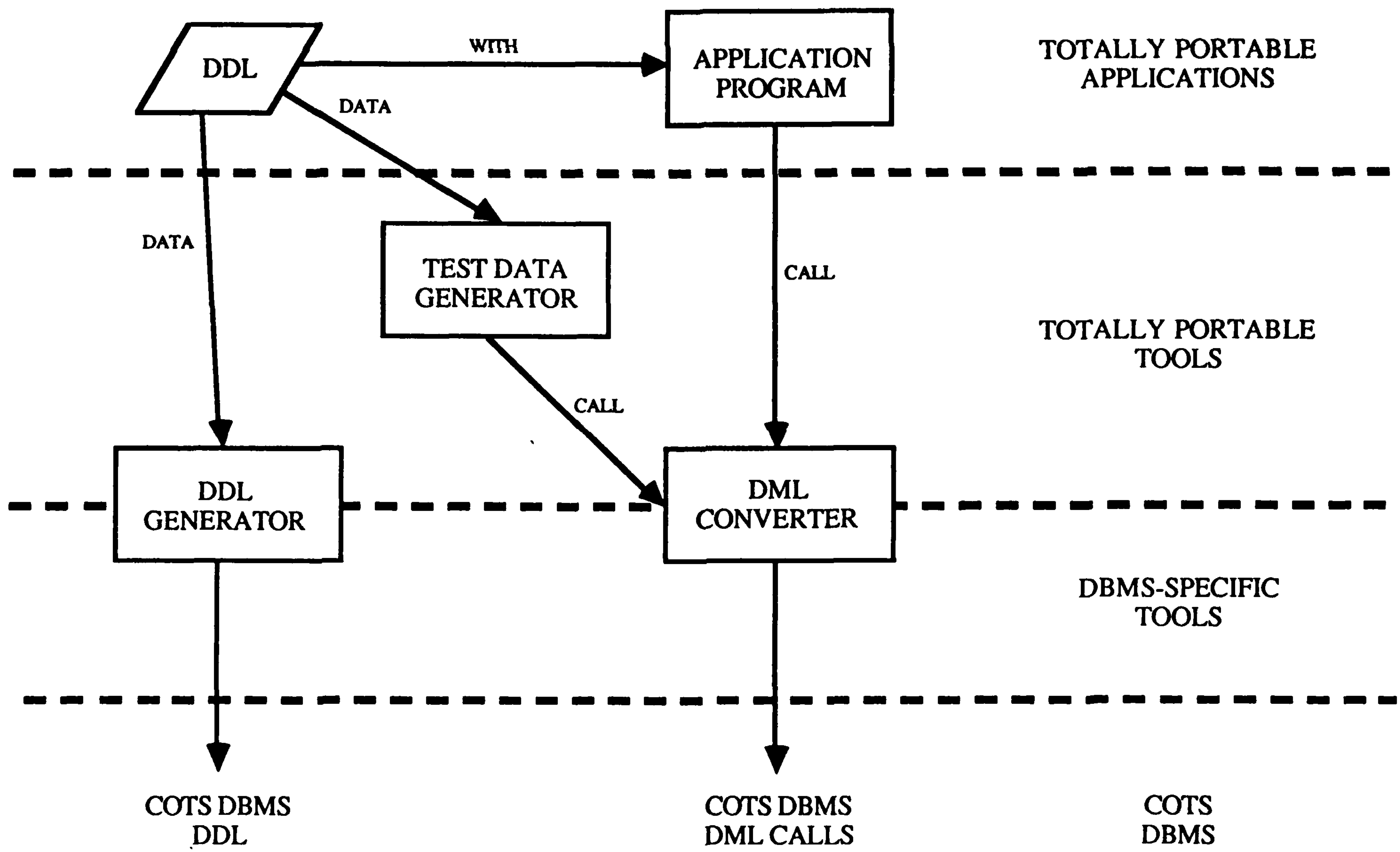


Figure 1. The Ada/SQL Interface Allows Totally Portable Applications and Tools

2.6. Data Definition Requirements

The main purpose of a relational data definition language is to define the tables that will be present within a database. Each table is named, and consists of one or more named columns. Each column has a particular data type for values that may be assigned to it. Unique keys for tables may also be specified, as this can be used both for consistency checking and for performance improvement. The Ada/SQL DDL includes provision for specifying all the above items, as well as privilege and view definition (these latter functions are not discussed here).

2.7. Automatic Test Data Generation Concerns

Additional information beyond that of a typical DDL is required for automatic generation of meaningful test data. Column data types are, of course, still required, but are now used to actively guide the data generated. In this regard, it is useful to know when the domain of one column is a subset of the domain of another. For example, EMPLOYEE and MANAGER columns may both be of type EMPLOYEE_NAME (a strongly typed version of STRING), but test data should be generated such that all MANAGERs are EMPLOYEEs, but only some EMPLOYEEs are MANAGERs.

Likewise, knowledge of key uniqueness is required to determine whether or not duplicate values should be generated for columns. In addition, which columns may be used for joins between tables must be known, so that corresponding data can be generated for join columns. The type of join for each pair of columns must also be known; whether the join is one-to-one, one-to-many, or many-to-many.

2.8. Features of the Ada/SQL DDL

The Ada/SQL DDL upholds the Ada philosophy of strong typing, by using Ada data types to also indicate database data types. Since the DDL is legal Ada that is "with'ed" into application programs, this naturally also determines the corresponding program data types. Strong typing also indicates which columns may be joined to which others, since join operations will only make sense between columns of the same data type. A typical COTS DBMS might only support one variety of STRING type, but the Ada/SQL DDL, Ada programs using it, and the test data generator will have the benefit of knowing about EMPLOYEE_NAMEs, HOME_ADDRESSes, etc. Strong typing will prevent a program from comparing an EMPLOYEE_NAME to a HOME_ADDRESS, and the test data generator will, of course, also generate different data for each type of column.

The Ada subtype mechanism provides a convenient technique for determining how data is to be subsetted for automatic generation. Expanding the EMPLOYEE and MANAGER example above, the Ada DDL statement

```
subtype MANAGER_NAME is EMPLOYEE_NAME;
```

makes it obvious that MANAGERs are a subset of EMPLOYEEs. The EMPLOYEE column would then be of subtype EMPLOYEE_NAME, while the MANAGER column would be of subtype MANAGER_NAME.

Type and subtype names may be suffixed with "_NOT_NULL" or "_NOT_NULL_UNIQUE" to indicate that columns of those types should have the appropriate SQL constraints. The test data generator will generate data in accordance with these constraints. For example, a column of type EMPLOYEE_NAME_NOT_ NULL_UNIQUE might be a key for an employee roster, and the test data generator would not generate any duplicate values for that column.

As noted above, column types determine which columns may be joined together. Thus, a column of subtype EMPLOYEE_NAME may be joined to one of subtype MANAGER_NAME, because both columns are of the same type, although of different subtypes. The uniqueness constraints given by the type and subtype definitions determine the type of join performed, and hence the nature of the test data generated. Joins between two unique columns are one-to-one, those between one unique and one non-unique column are one-to-many (the "one" side is obviously on the unique column), and joins between two non-unique columns are many-to-many.

Subsetting, as indicated by subtype, determines whether a join will match all values in a column or whether some values will not have corresponding matches in the other join column. There are three possibilities: (1) all values in each join column have corresponding value(s) in the other, (2) one join column has values that are a subset of the other, and (3) although some values in each join column overlap, both columns have values that are not present in the other. The Ada

DDL instructing the test data generator to produce data appropriate for each of these three cases is as follows:

```
(1) -- no special DDL required, both columns of subtype A

(2) subtype B is A; -- subset column of subtype B,
                    -- other column of subtype A

(3) subtype A is COMMON_TYPE;    -- one column of subtype A,
    subtype B is COMMON_TYPE;    -- other of subtype B
```

Two subtypes, one derived directly from the other and with names differing only in the "_NOT_NULL" or "_NOT_NULL_UNIQUE" suffixes, will not be subset, and are considered by the test data generator to be the same subtype for all purposes except uniqueness and the null value possibility. Thus, given the following DDL:

```
type EMPLOYEE_NAME is new STRING(1..20);
subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE
    is EMPLOYEE_NAME;
subtype MANAGER_NAME
    is EMPLOYEE_NAME_NOT_NULL_UNIQUE;
```

Columns of subtype EMPLOYEE_NAME and EMPLOYEE_NAME_NOT_NULL_UNIQUE will have the same range of values generated for them, except that null and duplicate values will be generated for EMPLOYEE_NAME columns but not for EMPLOYEE_NAME_NOT_NULL_UNIQUE columns. Only a subset of these values will be generated for columns of subtype MANAGER_NAME. The MANAGER_NAME subtype does not inherit the uniqueness or null constraints from EMPLOYEE_NAME_NOT_NULL_UNIQUE; the actual subtype name must include the suffix in order for the constraints to apply. The declaration for MANAGER_NAME is therefore equivalent, for test data generation purposes, to the following declaration:

```
subtype MANAGER_NAME is EMPLOYEE_NAME;
```

Database rows are indicated by Ada records in the Ada/SQL DDL, which is a most natural notation. Database columns are therefore represented, more or less, by the components of the Ada records. Ada records may, however, have components which are themselves records, and most COTS DBMSs do not support this subrecord concept. When encountering a subrecord, the DDL generator will expand it into its components, as many levels down as necessary, to produce the non-composite column descriptions required by the DDL of the target DBMS. Subrecords may be used advantageously in two ways: (1) they provide handy data types for items which are almost always processed together, such as latitude and longitude, and (2) they may be used to indicate uniqueness constraints over groups of columns (composite keys), even though each individual column within the group may not have unique values.

2.9. DDL Generator Portability Design

Reading and interpreting the Ada/SQL DDL is a non-trivial process -- trees of types and subtypes must be constructed; type definitions must be stored; records and their components must be noted, with type and name stored for each component; and enumeration literal values must be stored for each enumeration type. It is certainly desirable that it be as easy as possible to construct a DDL generator for a new COTS DBMS. For this reason, the DDL generator was designed to be as portable as possible across DBMSs. The Ada/SQL DDL is read and validated, with appropriate data structures built, in a completely DBMS-independent manner. Various backends can then be easily written (none written so far is as large as two pages of code) that use the data structures to produce the DDL required by a specific DBMS.

2.10. DDL Generator Example

Figure 2 shows the main program of a DDL generator that produces two different DDLs, one being a very simple DDL for a prototype DBMS and the other being for the DAMES DBMS, which has been interfaced to Ada. The routines of interest begin on line 21. Procedure SCAN_DDL performs all the chores of reading the Ada/SQL DDL and building the data structures describing the DDL read. As noted above, this is a fairly complex routine, but it is also totally DBMS-independent and portable. Although the DDL is written as an Ada package, our data structures do not have provision for storing the name of the package, so SCAN_DDL returns that to its caller, along with the index of the last character in the name string.

Procedure DISPLAY_DDL reformats the DDL processed into a listing which is useful to the DDL author. All DDL statements are "pretty printed", and trees showing the relationships of subtypes to types are printed in an indented, graphical format. This also is DBMS-independent.

Procedures GENERATE_SIMPLE_DDL and GENERATE_DAMES_DDL are the DBMS-specific routines that generate the DDL required by their respective DBMSs, using the database-independent data structures as input. (The DBMS-independent packages contain routines to access the data structures, so it is not necessary to pass the data structures as parameters.) As noted above, the DBMS-specific code required by these routines is less than two pages each.

```
with DAMES_DDL, READ_DDL, SHOW_DDL, SIMPLE_DDL, TEXT_IO,
        TEXT_PRINT, TOKEN_INPUT;
 use DAMES_DDL, READ_DDL, SHOW_DDL, SIMPLE_DDL, TEXT_IO,
        TEXT_PRINT, TOKEN_INPUT;

procedure MAIN is

  LINE         : LINE_TYPE;
  PACKAGE_NAME : STRING(1..80);
  LAST         : NATURAL;

  procedure PRINT_RULE is
  begin
    PRINT("--------------------------------------" &
          "--------------------------------------"); PRINT_LINE;
  end PRINT_RULE;

begin
  SET_STREAM(CREATE_STREAM(80)); OPEN_INPUT("BOATS.ADA");
  CREATE_LINE(LINE,79); SET_LINE(LINE);
  SCAN_DDL(PACKAGE_NAME,LAST);
  DISPLAY_DDL(PACKAGE_NAME(1..LAST)); PRINT_RULE;
  GENERATE_SIMPLE_DDL; PRINT_RULE;
  GENERATE_DAMES_DDL;
  CLOSE_INPUT;
end MAIN;
```

Figure 2. DDL Generators for Specific DBMSs are Easy to Write

2.11. Example of the Ada/SQL DDL

Figure 3 shows a sample database specification in the Ada/SQL DDL. The tables defined are a small part of an illustrative database used in [DATE 83]. As already stated, Ada record types are used to indicate the columns of database tables, as well as programming language subrecords within those columns. The PARCELS and PARCEL_ACCOUNTS record types define database tables, since they are not used as subrecords of any other records. On the other hand, the PARCEL_TRANSACTION_KEY record is used as a subrecord within another record, so the DDL generator will not produce a database table for it.

PARCEL_TRANSACTION_KEY_NOT_NULL_UNIQUE is used to designate a unique composite key (group of columns indicated by a subrecord) within the PARCEL_ACCOUNTS table. Another use of subtyping to indicate uniqueness is illustrated by the ASSESSOR_PARCEL_NUMBER_NOT_NULL_UNIQUE subtype. The example does not include a use of subtyping to show subsetting for test data generation.

In the example, all type definitions used for the database are included within a single package. This is not necessary; packages of type definitions may be "with'ed" into database definitions, as into any Ada packages. The Ada/SQL DDL also includes a facility for combining table definitions from several packages into a single database schema. Database definitions may therefore be organized in logical, modular fashion. Defining logically separate parts of a database in separate packages may reduce the number of application program recompilations required by changes to the data definition, since only users of

the affected packages must be adjusted. Each separate package may also be viewed as a subschema, defining and enabling operations on only a portion of the database. Views and privileges may also be used to define subschemas and protections for different classes of users of a database.

```
package ROAD_ASSOCIATION_SCHEMA is

      type ASSESSOR_PARCEL_NUMBER is new STRING(1..9);
      subtype ASSESSOR_PARCEL_NUMBER_NOT_NULL_UNIQUE
          is ASSESSOR_PARCEL_NUMBER;

   type ROAD_DESIGNATOR is (REDWOOD, CREEK, MILL);
   type OWNER_NAME is new STRING(1..20);
   type IMPROVED_FLAG is (Y,N);
   type ENTRY_NUMBER is range 1 .. 99999;
   type MONEY is delta 0.01 range -99_99.99 .. 99_999.99;
   type TRANSACTION_DESCRIPTION is new STRING(1..20);
   type CALENDAR_DATE is new STRING(1..6);
   type TRANSACTION_TYPE is (CHARGE, CREDIT);

   type PARCEL_TRANSACTION_KEY is
    record
       APN    : ASSESSOR_PARCEL_NUMBER;
       EN_TRY : ENTRY_NUMBER;
    end record;

   subtype PARCEL_TRANACTION_KEY_NOT_NULL_UNIQUE
       is PARCEL_TRANSACTION_KEY;

  type PARCELS is
    record
      APN        : ASSESSOR_PARCEL_NUMBER_NOT_NULL_UNIQUE;
      ROAD       : ROAD_DESIGNATOR;
      OWNER      : OWNER_NAME;
      IMPROVED   : IMPROVED_FLAG;
      LAST_ENTRY : ENTRY_NUMBER;
      BALANCE    : MONEY;
    end record;

  type PARCEL_ACCOUNTS is
    record
      KEY         : PARCEL_TRANSACTION_KEY_NOT_NULL_UNIQUE;
      DATE        : CALENDAR_DATE;
      DESCRIPTION : TRANSACTION_DESCRIPTION;
      TY_PE       : TRANSACTION_TYPE;
      AMOUNT      : MONEY;
      BALANCE     : MONEY;
    end record;
  .
   .
    .
end ROAD_ASSOCIATION_SCHEMA;
```

Figure 3. The Ada/SQL DDL is Pure Ada

2.12. Example of a Simple DDL Generated from the Ada/SQL DDL

Figure 4 shows the output of the GENERATE_SIMPLE_DDL procedure called from the main program previously shown, operating on the sample DDL just presented. The output is a DDL specification that can be read by a simple DBMS that had been prototyped earlier for the purpose of testing the Ada/SQL DML. (The DML was developed before the DDL; the prototype will eventually be modified to use the Ada/SQL DDL directly.) It can be seen how the OCEAN, ANALYST, SHIP, CREW, and SAMPLE_THIRD_LEVEL_RECORD records have become tables in the database; the subrecords have not. Also note, for example, how the LATLONG column in the SHIP record has been expanded into its component columns for the DDL. Other subrecord expansions are present in the example, including several levels' worth in the SAMPLE_THIRD_LEVEL_RECORD.

The translation of Ada types into database types is one of the functions of the DDL generator. For this particular DBMS, the translation is as follows:

```
Ada             DBMS
-----           --------

STRING(1..n)    STRING n   -- n = maximum number of characters
INTEGER         INTEGER 6  -- 6 = default print width supplied by generator
FLOAT           FLOAT 7    -- 7 = default print width supplied by generator
enumeration     STRING n   -- n = see below
```

The prototype DBMS uses the number following an INTEGER or FLOAT type declaration to indicate a print width for pretty printing the results of database queries. The default print widths currently supplied will be replaced with the appropriate computed widths when constraints are implement for numeric columns within the Ada/SQL DDL.

For this DBMS, enumeration values are store as strings, concatenating the print representation of their position with the literal representation for the value. Storing the position first enables the DBMS to sort them in the correct sequence, and storing the literal representation enables interactive users to see meaningful descriptions of column contents. The length of the string defined for a column corresponding to an enumeration type is therefore the sum of the number of digits required to represent the largest position plus the length of the longest literal value. For example, HUDSON_BAY is the OCEAN_NAME with the largest position value, which is 10. The longest OCEAN_NAME literal is 14 characters long (MEDITERRANEAN and GULF_OF_MEXICO), so columns of type OCEAN_NAME translate to STRING 16, 2 characters for the position and 14 for the literal representation. Leading zero's are used in the position to maintain the sort order, so the range of values for an OCEAN_NAME column runs from 01INDIAN to 10HUDSON_BAY. The NAME column of the OCEAN table and the OCEAN column of the SHIP table are examples of STRING 16 columns generated for OCEAN_NAME objects.

```
TABLE OCEAN
FIELD NAME STRING 16
FIELD ANALYST STRING 20
TABLE ANALYST
FIELD NAME STRING 20
FIELD SALARY FLOAT 7
FIELD MANAGER STRING 20
TABLE SHIP
FIELD NAME STRING 15
FIELD OCEAN STRING 16
FIELD LAT STRING 7
FIELD LONG STRING 8
FIELD TYPE STRING 10
TABLE CREW
FIELD TYPE STRING 10
FIELD SPECIALTY STRING 21
FIELD NUMBER INTEGER 6
TABLE SAMPLE_THIRD_LEVEL_RECORD
FIELD LAT STRING 7
FIELD LONG STRING 8
FIELD SCALAR_2 STRING 16
FIELD TYPE STRING 10
FIELD SPECIALTY STRING 21
FIELD SCALAR_3 INTEGER 6
END
```

Figure 4. A Simple DDL Generated from the Ada/SQL DDL

2.13. Example of DAMES DDL Generated from the Ada/SQL DDL

Figure 5 shows DDL for the DAMES database management system, as automatically generated by the program from the example Ada/SQL DDL. As interfaced to Ada, DAMES allows interactive program definition of tables, by calling the DEFINE_TABLE procedure. The DDL shown here is therefore legal Ada code, which may be copied into a program to perform the desired functions. The first parameter to DEFINE_TABLE is a STRING containing the name of the table to be defined, and the second parameter is a STRING containing the DDL defining the various columns of the table. The DAMES Ada interface supports the column data types STRING, INTEGER, and FLOAT. Also supported are enumeration types, the values of which must be explicitly listed for each column, and one level record types, the components of which must be explicitly listed for each column.

Generating DAMES DDL is more complex than generating the simple DDL previously discussed. For example, it is necessary to list the enumeration values for each enumeration type column. As can be seen from lines 13 and 30, these listings can be long, requiring continuation lines. Since STRINGs are being dealt with here, continuation lines are indicated by supplying the closing quotation mark on the current line, adding the string catenation operator to the line, then supplying an opening quotation mark for the start of the STRING on the next line. Although formatting such as this would ordinarily require a fair amount of code, many formatting functions have been designed into the totally transportable part of the DDL generator, thereby further simplifying the task of writing a DDL generator for a specific DBMS. The DAMES-specific code

prints enumeration values without concern for continuation lines, other than the one-time specification that a line is to be closed with a quotation mark and an ampersand before a continuation one is begun with another quotation mark. The standard, transportable code even handles indenting continuation lines.

Another complicating factor with DAMES is support for one-level subrecords. We have elected to retain the highest-level subrecords as DAMES subrecords, expanding all lower level subrecords. This decision was arbitrary: one could also expand all but the lowest level subrecords. The SAMPLE_THIRD_LEVEL_RECORD definition beginning on line 26 illustrates this point. It has a subrecord called SAMPLE_SECOND_LEVEL_RECORD, which, being the highest level subrecord, becomes a DAMES subrecord (line 27). The first component of SAMPLE_SECOND_LEVEL_RECORD is itself a subrecord, of type POSITION_LATLONG. Since DAMES only supports one level of subrecord, this subrecord had to be expanded into its constituent columns, LAT and LONG (lines 28 and 29). Had we decided instead to expand the lowest level subrecords, the components of SAMPLE_THIRD_LEVEL_RECORD would be (with a problem of duplicate column names):

```
FIRST_LEVEL_RECORD
  LAT
  LONG
SCALAR_2
FIRST_LEVEL_RECORD
  TYPE
  SPECIALTY
SCALAR_3
```

```
DEFINE_TABLE("OCEAN",
 "NAME (INDIAN,ATLANTIC,PACIFIC,MEDITERRANEAN,
     ARCTIC,CARIBBEAN," &
   "SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);" &
 "ANALYST STRING 1..20");

DEFINE_TABLE("ANALYST",
 "NAME STRING 1..20;" &
 "SALARY FLOAT;" &
 "MANAGER STRING 1..20");

DEFINE_TABLE("SHIP",
 "NAME STRING 1..15;" &
 "OCEAN (INDIAN,ATLANTIC,PACIFIC,MEDITERRANEAN,
     ARCTIC,CARIBBEAN," &
   "SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);" &
 "LATLONG " &
   "LAT STRING 1..7," &
   "LONG STRING 1..8;" &
 "TYPE (CARRIER,DESTROYER)");

DEFINE_TABLE("CREW",
 "KEY " &
   "TYPE (CARRIER,DESTROYER)," &
   "SPECIALTY (COOK,SHUFFLEBOARD_TEACHER);" &
 "NUMBER INTEGER");

DEFINE_TABLE("SAMPLE_THIRD_LEVEL_RECORD",
 "SECOND_LEVEL_RECORD " &
   "LAT STRING 1..7," &
   "LONG STRING 1..8," &
   "SCALAR_2 (INDIAN,ATLANTIC,PACIFIC,MEDITERRANEAN,
       ARCTIC,CARIBBEAN," &
     "SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);" &
 "FIRST_LEVEL_RECORD " &
   "TYPE (CARRIER,DESTROYER)," &
   "SPECIALTY (COOK,SHUFFLEBOARD_TEACHER);" &
 "SCALAR_3 INTEGER");
```

Figure 5. DAMES DDL Generated from the Ada/SQL DDL

2.14. The Ada/SQL Data Manipulation Language

This section of the report discussed the Ada/SQL data manipulation language (DML). An overview of the Ada/SQL DML, and its close resemblance to the SQL DML, is presented first. Next, examples of the Ada/SQL DML are provided. Finally, the implementation of the Ada/SQL DML is briefly discussed.

2.15. The Ada/SQL DML Looks Like SQL

Several representative statements from the Ada/SQL data manipulation language are shown in Figure 6. The underlying database in these examples is not the same as the one for the DDL example, rather it is taken from [DATE 83]. As part of developing the DML, all the functions necessary to process every example in the book were coded and executed, as verification of the versatility and completeness of the Ada/SQL DML.

It is surprising how close we can come to SQL using Ada syntax (remember, these examples are excerpts from actual programs that were compiled by a validated Ada compiler and then executed), but certain minor concessions did, of course, have to be made due to the natures of the two languages. For example, the SQL keywords SELECT and DECLARE are also reserved words in Ada, so the corresponding Ada/SQL subprograms are called SELEC and DECLAR. And, the arguments to the subprograms must have parentheses surrounding them (opening on line 1 and closing on line 9, for example), whereas no parentheses are used in SQL. The same naturally holds true for arguments to INSERT_INTO, UPDATE, and DELETE_FROM. The necessity to join separate words with underscores to make them single identifies in Ada is also apparent in CURSOR_FOR, GROUP_BY, ORDER_BY, INSERT_INTO, and DELETE_FROM.

Lists of items in SQL are separated by commas, in Ada/SQL they are separated by ampersands, since the ampersand can be overloaded as an Ada function whereas the comma cannot be. Examples of such lists can be seen in the following clauses: SELECT (line 2), FROM (line 3), GROUP BY (no example shown), ORDER BY (no example shown), INSERT INTO column list (line 22). For INSERT INTO value lists (line 24) the "and" operator is used as the connective. STRINGs would often be used as values, and using ampersands would have required redefining the array catenation operator, which is often used with STRINGs. UPDATE SET clauses (lines 27 and 28) are also separated by "and", in order to achieve the correct precedence between clauses. "<=" is used within each clause to indicate assignment of a value to a column, and an operator of lower precedence (i.e., not ampersand) must therefore be used to separate clauses.

The various SQL clauses become subprogram parameters in Ada/SQL. Thus, the clause names must be followed by the Ada parameter association symbol "=>", and the clauses must be separated by commas.

Perhaps the greatest concession in Ada/SQL was required by the restrictions on overloading the Ada equality and inequality operators ("=" and "/="). Ada/SQL functions return data structures, but these Ada operators can only be overloaded to return BOOLEAN, with operands of the same type. Hence, it is necessary for Ada/SQL to write "A=B" as EQ(A,B). This is required for comparison operators (see, for example, lines 4 and 5) and is the reason that "<=" is used instead of "=" for setting UPDATE values (see, for example, lines 27 and 28). Other SQL comparison and arithmetic operators that can be redefined in Ada are expressed in their natural fashion, however (line 8 shows an example of the greater than operator). SQL functions translate directly to Ada (see, for example, SUM on lines 2 and 8), but infix operators that have no equivalent in Ada must be written as Ada prefix functions (e.g., LIKE on line 6).

Several other minor concessions were also required. For example, an asterisk cannot stand by itself in Ada, as in "SELECT *" or "COUNT(*)", so the asterisk is instead made into a CHARACTER literal (line 15). Also, Ada strings are delimited by double quotes instead of the single quotes (apostrophes) used by SQL.

Even with all these concessions the similarity between SQL and Ada/SQL is remarkable. This is because the Ada language provides many features that can be exploited for Ada/SQL. Already noted were the use of subprograms and parameter names for SQL clause names, the direct translation of SQL functions such as SUM into corresponding Ada functions, and the redefinition of arithmetic and comparison operators other than equality and inequality. The redefinition of operators also applies to the boolean operators of "and", "or", and "not", so that predicates can be joined in their most natural fashion. The use of functions for Ada/SQL operators and SELECT statements allows nested queries, as with the EXISTS example beginning on line 11.

Database table and column names are also functions (defined by the SQL function generator from the DDL), overloaded to return objects of the appropriate type depending on the context in which they are used. The Ada record component selection operator (period) corresponds precisely with the SQL column selection operator, so qualified columns can be directly indicated in Ada (see, for example, line 17. PARCELS.APN is a column name qualified with a table name. The notation KEY.APN selects the KEY column in the PARCEL_ACCOUNTS table by SQL semantics, then the APN subcolumn by Ada semantics.)

```
A := SELEC    ( PARCELS.OWNER & SUM(PARCEL_ACCOUNTS.AMOUNT),
     FROM    => PARCELS & PARCEL_ACCOUNTS,
     WHERE   => EQ(PARCELS.APN,PARCEL_ACCOUNTS.APN)
     AND        EQ(PARCEL_ACCOUNTS.TYP,"CREDIT")
     AND        LIKE(PARCEL_ACCOUNTS.DATE,"82%"),
     GROUP   => PARCELS.OWNER,
     HAVING  => SUM(PARCEL_ACCOUNTS.AMOUNT) > 500,
     ORDER   => PARCELS.OWNER );

A := SELEC    ( APN & OWNER,
     FROM    => PARCELS,
     WHERE   => EXISTS
                ( SELEC   ( '*',
                  FROM  => PARCEL_ACCOUNTS,
                  WHERE => EQ(APN,PARCELS.APN)
                  AND      EQ(TYP,"CREDIT")
                  AND      AMOUNT > 499.99 ) ) );

A := INSERT_INTO ( PARCEL_ACCOUNTS ( EN_TRY & DATE & APN ),
          VALUES ( 99 and "850411" and "66-666-66" ) );

A := UPDATE ( PARCEL_ACCOUNTS,
     SET    => EQ(DESCRIPTION,"DUES82 TOO") &
               EQ(BALANCE,0.00),
     WHERE  => EQ(APN,"92-291-44")
     AND       EQ(DATE,"821212") );

A := DELETE (
     FROM   => PARCEL_ACCOUNTS,
     WHERE  => EQ(APN,"93-281-24")
     AND       EQ(AMOUNT,120.00) );
```

Figure 6. The Ada/SQL DML Looks Like SQL

2.16. The DML and DDL are Tied Together by Ada/SQL

The examples thus far did not include the use of program variables within SQL statements, but that is very straightforward. Any constant in the examples can obviously be replaced with a program variable; it is merely an argument to a subprogram. In short, program variables may be used anywhere they are semantically meaningful. There is no need (and no possibility, in fact) to differentiate program variables from database columns with a prefix such as the colon used by SQL. Of course, this does mean that program variables may not have the same names as database tables or columns, but this should not be a major problem. If record program variables are defined for each table in the database, using the types declared in the Ada/SQL DDL, then the record component names will be the same as the database column names. Continuing with the road association example, the program/PDL shown in Figure 7 illustrates the naturalness of this approach and the tie-in between the DDL and the DML.

The ROAD_ASSOCIATION_SCHEMA package "with"ed on line 1 is the Ada/SQL DDL already discussed, defining record types for the road association database. The ROAD_ASSOCIATION package on line 2 contains the definitions of all Ada/SQL functions (table and column names, and overloaded functions on unique data types) for the road association database. It is automatically generated from the Ada/SQL DDL by the SQL function generator. A use clause is not used for ROAD_ASSOCIATION_SCHEMA, defining record types, but is used for ROAD_ASSOCIATION, defining table names, because the automatic generation procedures cause table names to be homographs of database record type names. Table names are used in all Ada/SQL operations, while record type names are used only in declarations. Since the former use is expected to be more frequent than the latter, table names are made directly visible, while the record type names are visible only by selection. The SQL_OPERATIONS package on line 3 contains the Ada/SQL subprograms, such as SELEC, and operators not dependent on data types, such as the AND used to build search conditions.

Line 6 shows how the Ada/SQL DDL may be used within a program. PARCELS, in the ROAD_ASSOCIATION_SCHEMA package, is a definition of a record type used within the database. The CURRENT_PARCEL object of that type is the logical choice into which to retrieve tuples from the corresponding table. The CURSOR declared on line 7 is used with the DML to fetch successive tuples from a retrieved table. The exact fetch mechanism used is not shown here; it parallels the SQL FETCH-INTO operation.

Procedure SHOW_PARCELS_ON_ROAD asks the user to select a road (the PDL on line 9), queries the database for information on parcels on the selected road (the query is set up by the Ada/SQL on lines 10-13), and displays information on all parcels on the selected road (the PDL on lines 14-15 would turn into a loop in the actual code). Within the Ada/SQL query, PARCELS is the name of the database table generated from the PARCELS record type definition in the DDL. PARCELS.ROAD is the ROAD column in the PARCELS database table; CURRENT_PARCEL.ROAD is the ROAD component in the CURRENT_PARCEL program object. The way in which the functions and other declarations are set up causes this distinction to be automatically maintained by the Ada compiler.

```
 1. with ROAD_ASSOCIATION_SCHEMA;
 2. with ROAD_ASSOCIATION; use ROAD_ASSOCIATION;
 3. with SQL_OPERATIONS; use SQL_OPERATIONS;
 4.
 5. procedure SHOW_PARCELS_ON_ROAD is
 6.   CURRENT_PARCEL : ROAD_ASSOCIATION_SCHEMA.PARCELS;
 7.   CURSOR         : CURSOR_NAME;
 8. begin
 9.   read CURRENT_PARCEL.ROAD from user
10.   DECLAR   ( CURSOR , CURSOR_FOR =>
11.     SELEC  ( '*',
12.     FROM  => PARCELS,
13.     WHERE => EQ(PARCELS.ROAD, CURRENT_PARCEL.ROAD) ) );
14.   fetch successive database records into CURRENT_PARCEL, displaying
15.   information on parcels on the selected road to the user
16. end SHOW_PARCELS_ON_ROAD;
```

Figure 7. Relation of DDL and DML in Ada/SQL

2.17. Ada/SQL DML Implementation

The preceding discussion of Ada/SQL DML examples presented details of the strategy for implementing the SQL language within pure Ada code. This section concisely recapitulates the major ideas discussed.

Subprograms are defined for the basic SQL statements, such as SELECT. The parameters of these subprograms are given the same names as the SQL clause keywords, so that named parameter associations use the SQL clause keywords. Functions are also defined for those SQL operations, such as the AND used to build search conditions, that do not depend on database-unique data types.

Operations, such as EQ, that must be defined on user data types, are defined generically. The SQL function generator automatically generates instantiations of these functions based on the data types declared within the Ada/SQL DDL.

The SQL function generator also writes functions (most are generic instantiations) corresponding to database table names and column names. These functions are overloaded based on the type of result returned; the Ada compiler selects the correct version based on context. For example, a table name function returns a very simple data structure (an indication of the table name) when used within a FROM list. When used to qualify a column, however, as with PARCELS.APN, the table name function (PARCELS in this case) returns an access value designating a record object. The record object has one appropriately named component for each column in the corresponding table. Each component (such as APN) is a data structure describing both the table name and the column name.

2.18. Portability and Reuseability Considerations

This section of the report discusses the portability and reusability aspects of Ada/SQL. The SQL function generator is presented, along with issues relating to the DML converter.

2.19. SQL Function Generator Portability Concerns

The earlier picture of the Ada/SQL system did not show the SQL function generator, in the interest of simplicity. Its place in the total system is as shown in Figure 8. As discussed and shown, it reads the Ada/SQL DDL and produces a package defining functions for database tables, columns, and user-defined data types. This package is then "with'ed" into application programs using the database. The functions defined by the SQL function generator are independent of the underlying DBMS, so the SQL function generator is totally transportable.

2.20. SQL Function Generator Reusability Design

In the course of the Ada/SQL effort, the DDL generator was developed before developing the SQL function generator. (The DDL generator is really a family of programs, based on the target DDL, but it shall be referred to as a single program for ease of reference.) As it turned out, however, writing the SQL function generator was greatly simplified by the amount of code reused from the DDL generator. Both programs read the DDL file and produce textual output, and it has already been noted how the DBMS-independent part of the DDL generator performs the hard work of reading and validating the input DDL, and building data structures for subsequent processing. With these functions already in place, the SQL function generator code for translating the data structures into the textual package output was relatively straightforward.

2.21. Compile-time Checking of Operator Functions

Writing SQL within Ada enables the Ada compiler to perform type checking on database columns just as it does on program variables. This naturally improves the reliability and maintainability of the resultant programs.

Using the road association example again, the Ada compiler would not permit a programmer to say

```
EQ(PARCELS.ROAD, "I don't know")
```

for example. PARCELS.ROAD returns a data structure describing the database column selected. The type of this data structure is derived from the base type of all such data structures, specifically to correspond to the type of the column. Since the PARCELS.ROAD column is of type ROAD_DESIGNATOR, the SQL function generator might produce

```
type ROAD_DESIGNATOR_COLUMN is new COLUMN;
```

Code would then be generated to instantiate the EQ function, as well as all other appropriate ones, to allow comparisons between database columns and program variables of type ROAD_DESIGNATOR:

```
function EQ is new BINARY_OPERATOR(O_EQ,
  ROAD_DESIGNATOR_COLUMN, ROAD_DESIGNATOR_COLUMN);

function EQ is new BINARY_OPERATOR(O_EQ,
  ROAD_DESIGNATOR_COLUMN, ROAD_DESIGNATOR);

function EQ is new BINARY_OPERATOR(O_EQ,
  ROAD_DESIGNATOR, ROAD_DESIGNATOR_COLUMN);
```

BINARY_OPERATOR is a generic function for defining (not surprisingly) binary operators. Its first argument is an opcode (an enumeration type indicating the type of operation performed) to be placed into the data structure returned by the operator function. The next two arguments are the types of the left and right operands, respectively, of the binary operator. (This also applies to the two parameters, in order, of an operator, such as EQ, which must be written using Ada prefix function notation.) The first instantiation enables programs to compare two database columns of type

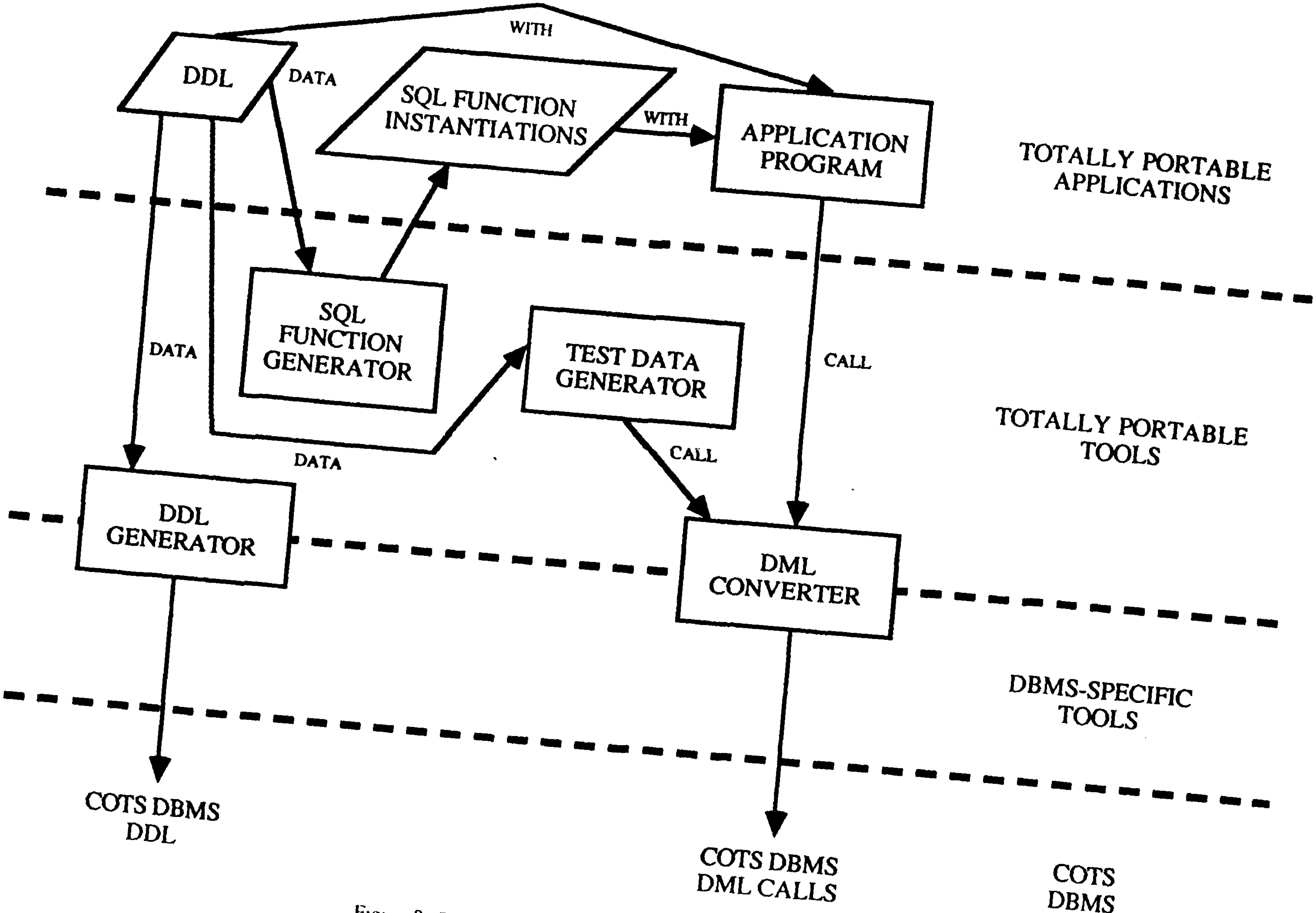


Figure 8. The Ada/SQL Function Generator is Totally Portable

ROAD_DESIGNATOR, which would most likely be used for joining tables. The remaining two instantiations enable programs to compare database columns with program objects and literals of type ROAD_DESIGNATOR. As a convenience, either order of the database column and the program object is permitted. As can be seen, there is no EQ function defined for comparing ROAD_DESIGNATOR_COLUMNs with string objects, which is why the Ada compiler would not permit the erroneous statement noted above.

Another function of operators such as EQ is to convert their program object operand (if any) to an internal representation that is independent of any user-defined data types. This is required so that the underlying Ada/SQL routines can operate on data types known to them; they do not know about any user-defined data types. (The underlying routines are independent of any specific database content.) The conversion is simple for integer, real, or string types -- values are just converted to the appropriate predefined type. Enumeration, array, and record types require data structures to represent values.

Much compile-time checking of SQL syntax is also provided by appropriately defining the Ada/SQL operators. If, for example, the EQ operator returned a result of type SEARCH_CONDITION, then the AND operator would be defined only for objects of type SEARCH_CONDITION. (Other definitions of "and" are provided for use within insert value lists and update set clauses, but they are not germane to this discussion.) This would make a use such as

```
WHERE => EQ(..) AND EQ(..)
```

legal, while

```
WHERE => "hello" AND "goodbye"
```

would be rejected by the compiler.

Similar syntax checking is applicable to lists of items. A typical example is the GROUP BY clause, which requires lists of column names. By defining the ampersand operator correctly, the compiler will accept

```
GROUP_BY => ROAD & OWNER
```

but will reject

```
GROUP_BY => ROAD & 7
```

2.22. DML Converter Portability Design

The DML converter is by far the most complex element of the Ada/SQL system. It takes the data structures built by the various Ada/SQL routines, such as SELEC, and translates them to the commands required by an underlying DBMS. There are, however, many functions that will be required by virtually all DBMSs, and these are implemented in a DBMS-independent manner for portability across DBMSs. Not all DBMSs will require each of these functions, so the DBMS-specific portion of any DML converter only calls those routines it requires.

For example, most DBMSs not implementing a SQL interface will require the DML converter to validate and process various aspects of SQL semantics. A good example of this is the qualification of unqualified columns. In SQL, column names may be used without specifying the names of the tables containing the columns, as long as the correct table can be uniquely determined. For processing SELECT statements, the code to make this determination is relatively complex, requiring checking the names of all columns in each table referenced in the FROM list of the query. If the column is referenced in a nested inner query, then FROM lists of successive outer queries must be checked until one or more tables containing the column is found. (Finding more than one such table in the same FROM list is an error, as is finding no such table after processing the outermost query.) Obviously, the DML converter for a specific DBMS is far simplified if all this checking is already available in a DBMS-independent manner.

Ada/SQL support for data types not supported by most COTS DBMSs also leads to another area in which portable functions are useful in the DML converter. As a simple example, the way in which enumeration values can be changed to strings has already been discussed. Conversions in both directions can be handled in a DBMS-independent manner, and called for those DBMSs that require them. A more complex example is the processing of subrecords. At the Ada/SQL level, for example, a single EQ operator compares all components of an Ada record to the values in what actually might be several database columns. This can be expressed in a DBMS-independent manner as a series of EQ operators, joined by ANDs, each comparing a record component to its corresponding database column. Again, DBMSs supporting subrecords would not

require this expansion of the query, but the code is available, in a portable manner, for those applications where it is required.

2.23. Summary

There are several objectives that are important for an Ada database interface. The interface should be portable by virtue of database independence, so that application programs using the interface can be run on any computer system using any underlying DBMS. Details of using a specific DBMS should be the concern of the interface implementation, not of the application programs. The interface should be written using pure Ada, in line with the philosophy of not subsetting or extending the language. It should be designed so that the resulting code satisfies the readability, reliability, and maintainability objectives of Ada itself.

Ada/SQL clearly satisfies these objectives. It also provides the additional advantage of being compatible with the SQL language, which is not only the most widely used relational database language, but which will also has been proposed as the ANSI standard language for relational databases.

3. Implementation Of Ada/SQL

This section of the report addresses the implementation of Ada/SQL. The Ada language design goals are presented, along with a discussion on how these goals are reflected in Ada/SQL. The implementation of the Ada/SQL DDL and DML is discussed at length. Next, several aspects of design considerations are presented, such as avoiding homographs, support for separate compilation, support for overloading, and support for strong typing considerations.

3.1. Ada Language Design Goals

The three major design concerns of Ada, as described within the Ada Language Reference Manual are:

```
o Program reliability and maintenance,

o Programming as a human activity, and

o Efficiency.
```

Each of these was also a design concern for Ada/SQL.

3.2. Ada/SQL Reliability and Maintenance

Considered foremost with respect to program reliability and maintenance was Ada's strong typing feature. Most programming languages and database interfaces support only classes of data types, permitting a variable or database column to be defined as containing integer, floating point, character (string) data, etc. Ada permits distinct data types to be defined within these classes (as well as within other classes), so that one particular string variable may be of type EMPLOYEE_NAME while another is of type EMPLOYEE_ADDRESS. Appropriate operations are defined for each data type, but they cannot be used with two different data types. Thus, a program can compare two objects of type EMPLOYEE_NAME, but it cannot compare an EMPLOYEE_NAME to an EMPLOYEE_ADDRESS. Since all variables and their types must be explicitly declared in Ada, this eliminates many errors caused by misspellings and other carelessness in other programming languages.

For Ada/SQL, Ada's strong typing was extended to relational databases. Using the data definition language, database columns can be defined to contain data of user-defined types such as EMPLOYEE_NAME and EMPLOYEE_ADDRESS. There are checks within the data manipulation language to ensure that operations are performed only on comparable objects. Thus, an EMPLOYEE_NAME database column can be compared with another EMPLOYEE_NAME database column or with an EMPLOYEE_NAME program value, but it may not be compared with a database column or program value of type EMPLOYEE_ADDRESS.

Since the Ada/SQL DDL is written using Ada, the necessary type definition facilities are available. A specific Ada construct, the record type definition, is used to indicate that a database table is being defined. The way that database columns are defined by record type definitions parallels the SQL table definition facility.

The DDL is compiled by the Ada compiler, so that all the type definitions are made available to programs using the database thereby defined. (Ada provides a separate compilation facility allowing type definitions and code from one compilation unit to be referenced from another.) The application programs may declare program variables of the appropriate types, and may also reference the database columns defined by the DDL. They include the necessary DML, which is also written using Ada, to operate upon the database. The direct Ada use of the DDL within programs containing DML plays a necessary part in implementing the strong type checking of Ada/SQL. The remaining part is played by automated tools that ensure consistency between the DDL and the DML.

Strong typing supports the reliability and maintenance of application programs. In addition, it supports total system reliability and maintenance. The Ada/SQL interface permits any (most likely relational) DBMS to be used to perform the actual database management function. Ada/SQL merely provides a consistent and transportable way of writing database commands in Ada; these operations are then translated to produce the desired results with the given DBMS. This translation function is another important aspect of the Ada/SQL software and automated tools.

Interfacing Ada/SQL to existing DBMSs provides the opportunity to use a mature, vendor-supported DBMS within a system. If database technology improves, if hardware changes, or for any other reason, a different underlying DBMS can be swapped into an Ada/SQL system, as long as the necessary Ada/SQL interface exists for the new DBMS. By virtue of Ada and Ada/SQL design goals, the application programs will be totally transportable to the new environment. Thus, systems are also not dependent on any particular hardware or DBMS vendor.

Using a DBMS also provides opportunities for enhancing data integrity Most existing systems (including SQL) enforce uniqueness and null value constraints. Systems are being developed and enhanced to also support referential integrity. Such integrity safeguards as are supported by the DBMS may, for the most part, be imposed upon the data without affecting the Ada/SQL application programs. (Certain multiple statement transactions may require delayed evaluation of the integrity constraints.) A program executing an operation that would cause an integrity constraint to be violated would receive an error indication from the DBMS; the operation would not be performed.

3.3. Ada/SQL Human Engineering

As already noted, the DDL and DML of Ada/SQL are both Ada. Programmers can therefore work completely within their Ada development environment, using whatever Ada tools are available to aid them. In particular, source code debuggers will display Ada/SQL data definition and manipulation statements precisely as coded, when used with an Ada/SQL development system.

Also as stated, the DDL and DML of Ada/SQL are as close to SQL as possible while still remaining Ada. SQL is by far the most mature of the database languages, is sanctioned by ANSI, and will undoubtedly continue to have the most DBMS products supporting it. Given all the design work and experience that have contributed to SQL, it was felt that it would be the easiest database language for Ada programmers to learn, as well as the one they would be most likely to already know.

As will be shown in the following implementation examples, the implementation of Ada/SQL was often faced with a trade-off between faithfulness to SQL and complexity of implementation. Where feasible, faithfulness to SQL at the expense of making implementation of the Ada/SQL interface more complex was preferred. The Ada/SQL interface is coded but once; programmers will continually be writing Ada/SQL programs.

A software system is usually written by more than one individual, and is typically built from several independent, but related, modules. Ada supports this by allowing data definitions and/or program routines to be gathered into separately compiled units called packages. Items defined in one package may then be used in other ones.

Consistent with this, Ada/SQL allows database definitions to be spread across several packages. If a definition within a package is changed, only those units depending on that package need be recompiled (or modified if required by the change). The Ada compiler environment keeps track of these package dependencies, so keeping data manipulation statements consistent with the definition of the data they manipulate is a natural consequence of Ada separate compilation concepts. Data manipulation statements comprising a system will, of course, typically be contained in many different separately compiled program units.

3.4. Ada/SQL Efficiency

Allowing various commercial DBMSs to be used with Ada/SQL rather than requiring a special-purpose, uniquely written data handler ensures that all data access efficiencies developed by the maturing technologies will be available to Ada/SQL programs. While this is certainly true in the traditional environment of a software DBMS on a single host computer, it is even more true with the emerging technologies of back-end and distributed database management systems. Ada/SQL programs will work equally well with these new DBMSs; the programs need not be aware of where the data are actually stored.

The implementation examples that will follow pertain to what is called a development mode Ada/SQL system. In a development mode implementation, all Ada/SQL statements are processed by the Ada compiler and executed as written, to allow display of the original program with source code debuggers. This requires that all database statements/transactions be built and processed at runtime, which can require more time and resources than will be acceptable in production applications.

When an application is ready for production, it can be processed by a production mode Ada/SQL system. In the production mode system, an automated tool preprocesses Ada/SQL data manipulation statements, replacing them with calls to commands that are designed for maximum database efficiency. Generating these database commands may include processing of the statements and transactions by the DBMS, so that optimum access paths and execution strategies can be devised before the program is even run. This obviously changes the form of the programs, which is the reason for also providing a development mode Ada/SQL system.

3.5. The Implementation of the Ada/SQL Data Definition Language

This section will examine how one part of SQL was implemented within Ada/SQL, for a development mode system. Recall that the strategy is to generate code from the DDL that will enable DML statements to compile. In discussing the code that will be generated, several stages will be passed through, to illustrate the design process required to achieve the final result. The first two strategies for generating code that will be shown are not actually used; they are simplified to aid in understanding of the ultimate implementation, which is quite complex. (The final strategy presented is actually also a simplification, but contains all the salient features of the actual implementation.)

In SQL, a table is defined by specifying its name (EMPLOYEE) and columns (NAME and STATUS) in a CREATE TABLE statement (see Figure 9). Each column has a data type, selected from a limited list of type classes, associated with it. In this example, both NAME and STATUS are character strings of the lengths indicated. Null and uniqueness constraints may also be specified for columns. In the EMPLOYEE table, NAME will be a primary key and so must contain unique, nonnull values.

In Ada/SQL, a table is defined by a record type declaration. The name of the record type being declared (again, EMPLOYEE, in this example paralleling the SQL) is taken as the name of the table being defined. An Ada record type permits several dissimilar items to be grouped into one object. Each of these items is called a component, and has both a name and a data type declared for it. The names of the components (NAME and STATUS) are taken to be the names of the columns within the table, similar to the SQL table definition statement. The types of the components are translated to the appropriate types for the columns as required by the underlying DBMS.

The Ada/SQL data types in the example are user-defined, and unique to each column according to the meaning of the data to be contained within the column. The NAME column is of type EMPLOYEE_NAME and subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE. The EMPLOYEE_NAME data type is obviously a string type of some sort: the complete definition of the type is not shown. In Ada, an object has both a type and a subtype. (Explicit declaration of a subtype is optional.) Subtype definitions permit constraints beyond those defined with the named type to be defined for objects of the subtype. In this example, the subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is used (by the Ada/SQL automated tools) to indicate that columns of that subtype are to be of type EMPLOYEE_NAME (as with Ada, from the subtype definition) and are to have the SQL NOT NULL UNIQUE constraint (from the suffix on the subtype name) applied to them.

The EMPLOYEE_STATUS data type used for the STATUS column is an enumeration type. A program variable (or a database column) of that type may contain only the values listed in its type definition -- HIRED, FIRED, or RETIRED, in this case. (They run a happy company; employees never quit.) These enumeration literals are used directly within Ada programs to represent the corresponding values, as shall be shown in the data manipulation example in Figure 10.

- SQL

```
CREATE TABLE EMPLOYEE
( NAME   CHARACTER(25) NOT NULL UNIQUE,
  STATUS CHARACTER(7) . . . )
```

- Ada/SQL

```
type EMPLOYEE_NAME is . . .
subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is EMPLOYEE_NAME;

type EMPLOYEE_STATUS is ( HIRED , FIRED , RETIRED );

type EMPLOYEE is
  record
    NAME   : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
    STATUS : EMPLOYEE_STATUS;
    . . .
  end record;
```

Figure 9. Data Definition Language

3.6. The Implementation of the Ada/SQL Data Manipulation Language

In SQL, data manipulation statements are specified as procedures, separate from the programs calling them, that are processed by the DBMS. (Provision is also made for embedding data manipulation statements within source programs, with minor syntactic variations.) In Figure 10 the example procedure is called RETIRE, since its function is to indicate that a given employee has retired. LEAVING_EMPLOYEE is a parameter to the procedure: its value is supplied as part of the program calling sequence to the RETIRE procedure. The calling sequence is not shown here. SQLCODE is a special parameter for the procedure; it returns a value within the calling sequence that is indicative of the completion status of the procedure.

The SQL UPDATE statement within the procedure contains three key words. Following the UPDATE key word is the name of the table to be updated, EMPLOYEE. The SET key word introduces a list of column names and associated values to place in those columns. This example sets the value in only one column, STATUS, to the string value 'RETIRED'. The WHERE key word precedes a search condition that restricts those rows in which the specified updates will be performed. For this example, a single row whose NAME column contains the value supplied for the LEAVING_EMPLOYEE parameter was selected. (No more than one row will be selected because NAME is a unique key. It is possible that no rows will satisfy the criterion.)

The corresponding Ada/SQL update statement is very similar. The example shown would be placed directly within the program using the statement; declaration of a separate procedure is not necessary. LEAVING_EMPLOYEE_NAME is the program variable that will contain the name of the retiring employee; it is naturally declared to be of type EMPLOYEE_NAME. No special indication is required within the Ada/SQL update statement to show that LEAVING_EMPLOYEE_NAME is a program variable rather than a database column name; the distinction is maintained according to the declarations supplied by the programmer vs. those generated by the Ada/SQL system from the DDL. A special SQLCODE status indicator is also not necessary; errors are signaled via the usual Ada exception mechanism. Several functions are available to enable application programs to obtain more detailed information about errors after exceptions are raised.

UPDATE is a procedure with three parameters. The name of the procedure places the UPDATE key word into the statement, which is actually a call to the UPDATE procedure. Subprogram parameters are named in Ada, and the names may be used in the call, followed by the "=>" symbol, to indicate values for the parameters. Hence, the second and third parameters to the UPDATE procedure are named SET and WHERE, in order to get those keywords into the Ada/SQL statement. The value for the first parameter defined for the procedure, the table to be updated, is supplied by the first actual parameter (EMPLOYEE) used in the call. Ada calls this positional association, as opposed to the named association used for

the second and third parameters.

The value of the SET parameter to the UPDATE procedure, in this example, is the result of calling a function, "<=", with two parameters, STATUS and RETIRED. "<=" is the operator symbol used by Ada to indicate "less than or equal to", but in this context the intended meaning is to be "gets". Ada provides the facility to define new meanings for many of its operators. We will see the details of this in Figure 10, but suffice it to say for now that Ada makes it possible to define a new meaning for "<=" in this context, while retaining the expected meaning of "less than or equal to" in other contexts.

STATUS is a call to a function of that name, with no parameters, that is defined within the code automatically generated from the DDL declaring the STATUS column within the EMPLOYEE table. (EMPLOYEE and NAME, used elsewhere within the statement, are similar functions.) RETIRED is a value of the EMPLOYEE_STATUS enumeration type; it is a program literal with a constant value.

The value of the WHERE parameter is similarly the result of calling the EQ function with parameters being (1) the result of calling the NAME function and (2) the value of the LEAVING_EMPLOYEE_NAME program variable. Positional association is used for the parameters to the EQ function. It would have been desirable to use the same infix operator notation used with the "<=", so that the WHERE parameter could be written as

```
NAME = LEAVING_EMPLOYEE_NAME
```

but Ada places certain restrictions on redefining the "=" operator that prevent us from using this notation for equality. The declarations of the UPDATE, "<=", and EQ functions are all automatically generated from DDL shown in Figure 10. Now the details of implementing the "<=" function will be shown as an example of how the necessary functions may be defined to permit Ada/SQL statements to be compiled by the Ada compiler. Remember that this will build up to the final implementation strategy by first presenting two illustrative strategies that are simpler but not totally satisfactory.

- SQL

```
PROCEDURE RETIRE
     LEAVING_EMPLOYEE CHARACTER(25)
    SQLCODE;
    UPDATE EMPLOYEE
    SET    STATUS = 'RETIRED'
    WHERE  NAME = LEAVING_EMPLOYEE;
```

- Ada/SQL

```
LEAVING_EMPLOYEE_NAME : EMPLOYEE_NAME;
. . .
UPDATE ( EMPLOYEE,
SET    => STATUS <= RETIRED,
WHERE => EQ ( NAME , LEAVING_EMPLOYEE_NAME ) );
```

Figure 10. Data Manipulation Language

3.7. The Implementation of Assignment Operators

As mentioned, STATUS is a function with no parameters. In Ada, the types of all subprogram parameters must be declared, as must the types of all values returned by functions. (There are two types of subprograms, procedures and functions. Functions return a value, procedures do not. Any subprogram may have no parameters.) Based on the DDL, one can generate the definition of a STATUS function returning a value of type COLUMN_NAME, since STATUS is, in fact, used as a column name in the context being discussed.

What is this type COLUMN_NAME? It is a data structure used internally within the Ada/SQL interface. It must clearly include at least an indication of which column is being named, so that the fact that STATUS was used can be reconstructed. The application program calling STATUS need have no knowledge of the internal structure of a COLUMN_NAME object, however. In fact, we certainly don't want application programs to be able to manipulate values internal to Ada/SQL data

structures, since they may thereby be able to subvert system integrity and negate any advantage of strong typing.

The Ada private type facility precisely supports this requirement. A private type is defined within a package. (Recall that a package is a separately compiled program unit that may define both types and subprograms.) Subprograms defined within that same package may perform any operations on objects of a private type that would be permitted had the type not been private. Subprograms defined elsewhere, however, may only perform operations that are authorized by the package defining the private type. They therefore have restricted and controlled capability to manipulate objects of a private type.

COLUMN_NAME is defined as a private type within the Ada/SQL definitions. The internal Ada/SQL routines defined within the same package can perform all required operations on COLUMN_NAME objects. Application programs, however, may only use certain authorized operations on COLUMN_NAME objects. These include calling subprograms explicitly made available to them with parameters and/or returned values of type COLUMN_NAME, such as the STATUS and "<=" functions defined here. Just as application programs need not know the inner workings of COLUMN_NAMEs and other private types, it is also sufficient for us to know what subprograms are generated for them in order to understand how the generated subprograms fit together to enable Ada/SQL statements to compile. We will therefore not give any precise details on the internal structure of any private types.

Since the STATUS column is of type EMPLOYEE_STATUS, a "<=" function is generated that permits a COLUMN_NAME as the left parameter (the value returned by STATUS) and an EMPLOYEE_STATUS program value (such as the literal RETIRED) as the right parameter. What is the type of the value returned by this "<=" function?

This particular "<=" function is used in Ada/SQL contexts corresponding to SQL set clauses, so the data structure returned by it is called a SET_CLAUSE. Again, SET_CLAUSE is a private type, with details known only to the Ada/SQL interface internal routines. Information placed within the SET_CLAUSE object returned by the "<=" function defined here includes the name of the column to be updated (taken from the COLUMN_NAME left parameter) and the value to be used (taken from the EMPLOYEE_STATUS right parameter).

A function defined in this way, with an operator symbol as its name, may be called using the more natural infix notation, as in

```
STATUS <= RETIRED,
```

thus implementing the desired feature. One other "<=" function must also be generated for columns of type EMPLOYEE_STATUS, since the updating value may not be a simple program value of type EMPLOYEE_STATUS, such as RETIRED. It may instead be an expression involving database columns. The operators used to build such an expression return data structures indicating all the operators and operands used. Such data structures are, of course, of private types, and further description of them is beyond the scope of this presentation.

Other "<=" operators (functions) are generated to accept right parameters of the other user-defined data types used for database columns. This will be discussed shortly, but first a question is posed: How many "<=" functions can you have, anyway? The answer requires an explanation of the Ada overloading concept.

Most programming languages, even those without strong typing, support different versions of the same operator. For example, A+B typically invokes integer addition if both A and B are integers, floating point addition if both A and B are floating point, and a conversion from integer to floating point followed by floating point addition if one operand is integer (it is the one converted) and one is floating point. Since the parameters and (for functions) return values for Ada subprograms are all strongly typed, the Ada compiler can determine which of several identically-named subprograms is being called according to the types of the actual parameters supplied and (for functions) the expected type of the result. In some cases this overloading resolution can get extremely complex, but a simple example will readily demonstrate it.

It was noted before that "<=" is also the "less than or equal to" comparison operator. One certainly may want to compare the value in a database column with the value of a program expression, as in (is) STATUS <= RETIRED (?). (The values of an enumeration type are ordered the same as they are declared. In this example, HIRED is less than FIRED, and all EMPLOYEE_STATUS values are less than or equal to RETIRED.) Therefore a "<=" operator must be generated for this purpose. Although this operation appears the same as the one in Figure 10, it will be used in different contexts. The private type data structure used for those contexts happens to be called SEARCH_CONDITION, corresponding to the SQL syntactic context. This "<=" operator takes parameters of the same type as the one previously discussed (repeated here for convenience), but returns a result of a different type:

```
function "<=" ( LEFT  : COLUMN_NAME;
                RIGHT : EMPLOYEE_STATUS ) return SET_CLAUSE;

function "<=" ( LEFT  : COLUMN_NAME;
                RIGHT : EMPLOYEE_STATUS ) return SEARCH_CONDITION;
```

(Wherever the definition of a subprogram is shown, the Ada specification for the subprogram is given. Ada permits the specification of a subprogram to be written separately from its body. The specification provides the calling sequence for the subprogram, including the name of the subprogram, the name and type of each parameter, and, for a function, the type of the returned result. An indication of whether a subprogram parameter is input, output, or both is also part of the specification, and will be shown for procedure specifications. Parameters to functions are restricted by Ada to be input only. In the second example above, the subprogram name is "<=", it is a function returning a result of type SEARCH_CONDITION, its first parameter is named LEFT and is of type COLUMN_NAME, and its second parameter is named RIGHT and is of type EMPLOYEE_STATUS. The body of a subprogram contains the actual code that is written to implement its operations. Knowing the specifications of the various Ada/SQL functions is sufficient to understand how Ada capabilities are used to implement software engineering concepts within Ada/SQL. The details of the subprogram bodies are not shown here.)

With TABLE_NAME being another private type with obvious connotation, the UPDATE procedure can now be completely declared in Figure 10 as:

```
procedure UPDATE ( TABLE  : in TABLE_NAME;
                   SET    : in SET_CLAUSE;
                   WHERE  : in SEARCH_CONDITION );
```

If the call were instead (a relatively useless operation):

```
UPDATE ( EMPLOYEE,
SET   => STATUS <= RETIRED,
WHERE => STATUS <= RETIRED );
```

then the value of the SET parameter is computed by calling the "<=" function returning type SET_CLAUSE, while the value of the WHERE parameter is computed by calling the "<=" function returning type SEARCH_CONDITION. These are the types expected by the UPDATE procedure, and Ada overloading resolution causes the correct functions to be called. The code for each overloaded subprogram is independently defined, and can be quite different even though the names of the subprograms are the same.

3.8. Lack of Support for Strong Typing

Just as the STATUS column causes the following function to be automatically generated from the DDL:

```
function STATUS return COLUMN_NAME;
```

so must the NAME column also cause the following function to be generated from the DDL:

```
function NAME return COLUMN_NAME;
```

The fact that the STATUS column was of type EMPLOYEE_STATUS required the following function to be generated:

```
function "<=" ( LEFT  : COLUMN_NAME;
                RIGHT : EMPLOYEE_STATUS ) return SET_CLAUSE;
```

Likewise, since the NAME column is of type EMPLOYEE_NAME (it is of subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE, but its base type is EMPLOYEE_NAME), it must also generate:

```
function "<=" ( LEFT  : COLUMN_NAME;
                RIGHT : EMPLOYEE_NAME ) return SET_CLAUSE;
```

Since the STATUS function returns an object of type COLUMN_NAME, it might be used as the left argument to this "<=" function, thereby permitting one to write an assignment of an EMPLOYEE_NAME value to the STATUS column, an operation that must not be permitted according to the strong typing philosophy:

```
LEAVING_EMPLOYEE_NAME : EMPLOYEE_NAME; -- a program variable,
                                       -- as before
. . .
UPDATE ( EMPLOYEE,
SET    => STATUS <= LEAVING_EMPLOYEE_NAME . . . );
```

Similarly, functions have now been defined to permit an EMPLOYEE_STATUS value to be assigned to the NAME column, so the following erroneous operation could also be coded:

```
UPDATE ( EMPLOYEE,
SET    => NAME <= RETIRED . . . );
```

How can the code generated be augmented from the DDL to support the strong typing that will prevent the coding of such nonsensical assignments?

3.9. Support for Strong Typing through Derived Types and Overloading

The problem with the above implementation strategy is that there is nothing in the COLUMN_NAME data type returned by the STATUS function to indicate that the STATUS column contains data of type EMPLOYEE_STATUS. What is desired, therefore, is to have a special version of the COLUMN_NAME data type that is specific for columns containing data of type EMPLOYEE_STATUS. This data type would be called EMPLOYEE_STATUS_COLUMN_NAME, formed by adding the _COLUMN_NAME suffix to the user data type name. Similarly, there would be an EMPLOYEE_NAME_COLUMN_NAME type to be returned by the NAME function, for the NAME column of base type EMPLOYEE_NAME.

The EMPLOYEE_STATUS_COLUMN_NAME and EMPLOYEE_NAME_COLUMN_NAME types must be distinct, so that operations on one type will not apply to the other type, but their data structures must also contain the same information as the basic COLUMN_NAME data structure. Ada provides the perfect construct for implementing this, called derived types. One may derive types EMPLOYEE_STATUS_COLUMN_NAME and EMPLOYEE_NAME_COLUMN_NAME from type COLUMN_NAME with:

```
type EMPLOYEE_NAME_COLUMN_NAME   is new COLUMN_NAME;
type EMPLOYEE_STATUS_COLUMN_NAME is new COLUMN_NAME;
```

Each new type is distinct, but has the exact same data structure as does type COLUMN_NAME. Because the new types are distinct, a subprogram expecting a parameter of type EMPLOYEE_NAME_COLUMN_NAME will not accept one of type EMPLOYEE_STATUS_COLUMN_NAME, and vice versa. However, subprograms that know that (for example) type EMPLOYEE_STATUS_COLUMN_NAME is derived from type COLUMN_NAME may convert values from one type to the other. (The conversion does not actually require any code at runtime, since the data structures are identical. An explicit conversion syntax is required within the program, however, as part of Ada's regimen of strong typing.) Ada/SQL routines generated from the DDL may therefore accept parameters of type EMPLOYEE_STATUS_COLUMN_NAME and convert them to the standard internal COLUMN_NAME type for processing by the inner layers of Ada/SQL software that are not dependent on the database defined by the user.

Application programs do not know that all these type-specific column name data types are derived from the same parent COLUMN_NAME data type. If they did, they would be able to explicitly convert, for example, a value of type EMPLOYEE_NAME_COLUMN_NAME to type EMPLOYEE_STATUS_COLUMN_NAME, which would enable them to subvert the strong typing mechanism which is being built. Ada's private type facility, already discussed, is used to hide these derivations from the application programs.

The STATUS function, used to reference the STATUS column of the EMPLOYEE table, must therefore return an object of type EMPLOYEE_STATUS_COLUMN_NAME, indicating that its syntactic and semantic use is as the name of a column containing data of type EMPLOYEE_STATUS. To accomplish this, the following function is generated from the DDL defining the STATUS column:

```
function STATUS return EMPLOYEE_STATUS_COLUMN_NAME;
```

The type-specific "<=" function used to assign values to the STATUS column (actually to any column containing data of type EMPLOYEE_STATUS) generated from the DDL is:

```
function "<" ( LEFT  : EMPLOYEE_STATUS_COLUMN_NAME;
               RIGHT : EMPLOYEE_STATUS ) return SET_CLAUSE;
```

It is this function that is called when the RETIRED value is assigned to the STATUS column with STATUS <= RETIRED. The STATUS function returns the EMPLOYEE_STATUS_COLUMN_NAME type left parameter to the "<=" function, while the RETIRED program literal of type EMPLOYEE_STATUS provides the right parameter to the "<=" function. Similar functions for the NAME column are generated as follows:

```
function NAME return EMPLOYEE_NAME_COLUMN_NAME;

function  <=" ( LEFT  : EMPLOYEE_NAME_COLUMN_NAME;
                RIGHT : EMPLOYEE_NAME ) return SET_CLAUSE;
```

Note that a violation of strong typing, such as NAME <= RETIRED, will not now compile, since there is no "<=" function defined for that combination of column name and program value. The required function would have a left parameter of type EMPLOYEE_NAME_COLUMN_NAME and a right parameter of type EMPLOYEE_STATUS. Such a function would never be generated from the DDL, as is obvious from the scheme presented here.

3.10. Lack of Support for Separate Compilation

It has been noted before that the Ada package concept permits a program to be segmented into several separately-compiled units. This facility is, naturally, supported by Ada/SQL. Suppose one desires to define two tables of employee information. The EMPLOYEE table, whose definition has already been seen, will contain current information, with a single row for each employee. The EMPLOYEE_HISTORY table, on the other hand, will contain historical information, so that more than one row may be included for a given employee. Many programs will use one table but not the other, so it is natural to define each table in a separate package. If the definition of one table changes, it will not be necessary to recompile programs using only the other table. Again, these recompilation dependencies are managed by the Ada language system. Changing a table definition is accomplished by changing a DDL package, and the Ada compiler will require recompilation of all program units dependent on the modified package.

Since data of the same types will be stored in both the EMPLOYEE and the EMPLOYEE_HISTORY tables, still another package will be used to define those data types (the definitions are the same as for the previous example):

```
package A_SCHEMA is
  type EMPLOYEE_NAME is . . .
  subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is EMPLOYEE_NAME;
  type EMPLOYEE_STATUS is ( HIRED , FIRED , RETIRED );
end A_SCHEMA;
```

The EMPLOYEE table may then be defined, as before, in its own package. In this and other examples, context clauses that are required by the Ada compiler to indicate how definitions from other packages are to be used within a given package are not shown. The examples shown thus far are simple enough that it is clear which definitions are being used:

```
package B_SCHEMA is
  type EMPLOYEE is
  record
    NAME   : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
    STATUS : EMPLOYEE_STATUS;
    . . .
  end record;
end B_SCHEMA;
```

And the EMPLOYEE_HISTORY table is similarly defined, in its own package. Note that the NAME column is not a unique key in the EMPLOYEE_HISTORY table. (To be rigorous, a NOT NULL constraint should probably be defined on the NAME column, but that degree of detail is not necessary in this example.)

```
package C_SCHEMA is
  type EMPLOYEE_HISTORY is
  record
    NAME   : EMPLOYEE_NAME;
    STATUS : EMPLOYEE_STATUS;
    . . .
  end record;
end C_SCHEMA;
```

For the implementation strategy now being discussed, the type-specific types derived from type COLUMN_NAME and the "<=" operator functions are dependent solely on user types defined within the DDL from which they are generated; they are independent of any tables defined. Consequently code for those items is generated in packages corresponding to those in which the user types are defined. This results in only a single type-specific column name type, such as EMPLOYEE_STATUS_COLUMN_NAME, being derived for each user-defined type, no matter how many other packages define tables containing columns of the user-defined type. This naturally also limits the number of "<=" and similar functions that must be generated. (It will be shown later that the types derived from type COLUMN_NAME and their associated functions must also be made dependent on the tables in which columns of the corresponding user-defined types are used, which will require the generated definitions to be moved into the packages generated for the packages in which the tables are defined. There are, however, many other types and functions within Ada/SQL that are dependent only on the user types defined, and their placement within the generated packages is as described here.) The package generated to correspond to A_SCHEMA, which contains the user type definitions, is therefore:

```
package A is
  type EMPLOYEE_NAME_COLUMN_NAME is new COLUMN_NAME;
  type EMPLOYEE_STATUS_COLUMN_NAME is new COLUMN_NAME;
  function "<=" ( LEFT   : EMPLOYEE_NAME_COLUMN_NAME;
                  RIGHT  : EMPLOYEE_NAME ) return SET_CLAUSE;
  function "<=" ( LEFT   : EMPLOYEE_STATUS_COLUMN_NAME;
                  RIGHT  : EMPLOYEE_STATUS ) return SET_CLAUSE;
  . . .
end A;
```

These definitions are the same as have already been shown generated for the EMPLOYEE_NAME and EMPLOYEE_STATUS user-defined data types. Note that the names of packages defining Ada/SQL DDL are given the suffix _SCHEMA by the users creating them. Corresponding packages generated from the DDL packages are given the same names, without the _SCHEMA suffix. (The names of all packages defining database tables must have the _SCHEMA suffix. The names of packages defining types used within database tables, but not any database tables, need not end with the _SCHEMA suffix. A different naming convention, which is not relevant to this discussion, is then used to name the corresponding generated packages.)

The functions for referencing the names of columns defined for the EMPLOYEE and EMPLOYEE_HISTORY tables are then generated in packages corresponding to those in which the tables are defined:

```
package B is
  function NAME return EMPLOYEE_NAME_COLUMN_NAME;
  function STATUS return EMPLOYEE_STATUS_COLUMN_NAME;
  . . .
end B;

package C is
  function NAME return EMPLOYEE_NAME_COLUMN_NAME;
  function STATUS return EMPLOYEE_STATUS_COLUMN_NAME;
  . . .
end C;
```

Again, these are definitions that have been shown before. There are, of course, other functions generated for other uses of table and column names: these are indicated by the ellipses. Note that each package defines distinct functions, even though the definitions appear to be identical. In Ada, the notation B.NAME is used to indicate the NAME function defined in package B, C.NAME indicates the NAME function defined in package C, etc. The ability to use identical names for

different things in different packages is an important software engineering concept of Ada -- it means that packages may truly be written independently, without concern for names used in other packages.

3.11. Homographs

There is, however, a problem with having, for example, two identical STATUS functions. Assume that the application program is using the definitions in both packages B and C. In the update of the STATUS column,

```
UPDATE ( EMPLOYEE,
  SET => STATUS <= RETIRED . . . );
```

the Ada compiler cannot determine whether STATUS is intended to be a call to the STATUS function defined in package B (B.STATUS) or the STATUS function defined in package C (C.STATUS), since those functions have identical definitions. It is obvious that the STATUS function of package B is intended, since the EMPLOYEE table is defined in B_SCHEMA. But, some way of also letting the Ada compiler know this too is needed.

The most obvious way to do this is to use Ada's selected component syntax to explicitly tell the compiler which STATUS function to use:

```
UPDATE ( EMPLOYEE,
  SET => B.STATUS <= RETIRED . . . );
```

It was felt that this was not an acceptable solution, however, since the "B." prefix has no semantic meaning in SQL. It arises solely from the Ada program package structure, and has absolutely no counterpart in SQL. Modified SQL syntax has been shown to build it into Ada has been shown, but nowhere (except for one construct where SQL violates important software engineering principles) has it altered SQL semantics to create Ada/SQL.

SQL does provide the facility to qualify a column name with the name of its table, as in EMPLOYEE.STATUS for the STATUS column within the EMPLOYEE table, and it would be possible to generate functions so that the following version of the update statement would compile correctly:

```
UPDATE ( EMPLOYEE,
  SET => EMPLOYEE.STATUS <= RETIRED . . . );
```

In fact, the EMPLOYEE.STATUS notation is used elsewhere within Ada/SQL with its same meaning as in SQL -- the STATUS column of the explicitly-specified EMPLOYEE table. (It is fortunate that the period has a meaning in Ada that permits this exact same notation as in SQL. Further description of that aspect of Ada/SQL implementation is not provided here.)

But, again, this version of the update statement is not acceptable. SQL does not permit the name of a column being updated to be prefixed with the name of its table. Semantically, all columns to be updated must be contained within the table being updated. The Ada/SQL implementation should support the form of the update statement closest to SQL, as already presented:

```
UPDATE ( EMPLOYEE,
  SET => STATUS <= RETIRED . . . );
```

3.12. Support for Separate Compilation through Derived Types and Overloading

Recall that the EMPLOYEE table reference in the update statement is actually a call to a function that returns an object of type TABLE_NAME. Suppose instead that its return value has been typed to specifically indicate that the EMPLOYEE table was referenced:

```
type EMPLOYEE_TABLE_NAME is new TABLE_NAME;

function EMPLOYEE return EMPLOYEE_TABLE_NAME;
```

Here, the type name is constructed by appending the _TABLE_NAME suffix to the table name. Ada language

considerations for the use of derived types for table names are as already discussed for column names.

One can now have each function implementing a column reference return a value typed specifically to the table in which the column is defined. Each column name derived type is already dependent on the data type of the column. Dependence is now being added on the column's table, so the type name is created by taking the table name (e.g., EMPLOYEE), appending an underscore, appending the name of the data type contained within the column (e.g., EMPLOYEE_STATUS), and finally appending the _COLUMN_NAME suffix. A typical type definition now looks like:

```
type EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME is new COLUMN_NAME;
     --------------    -----------------    -----------
     table             column               suffix for
                       data type            syntactic use
```

There are certain combinations of table and user type names that can provide duplicate type names. For example, a table named EMPLOYEE_EMPLOYEE containing a column of type STATUS would also give rise to the above type definition, which was also created for the table named EMPLOYEE containing a column of type EMPLOYEE_STATUS. Such potential duplication is not a problem for Ada/SQL, since these type names are never actually generated. (Remember that even the complex ultimate implementation will be simplified for ease of presentation.) Although distinct types dependent on table and column data type are generated, as conceptually presented here, they are in fact implemented within generic packages in a fashion such that Ada's package facility keeps name conflicts from arising.

The complete definitions of the table- and type-specific column name types and the column reference functions for the EMPLOYEE table are:

```
type EMPLOYEE_EMPLOYEE_NAME_COLUMN_NAME   is new COLUMN_NAME;
type EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME is new COLUMN_NAME;

function NAME   return EMPLOYEE_EMPLOYEE_NAME_COLUMN_NAME;
function STATUS return EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME;
```

Before, the types derived were from COLUMN_NAME defined in the package generated from the one in which the various user data types were defined. This was fine as long as the derived types were dependent only on the user data types. Now that each derived type is also dependent on the table in which the data type is used, the definitions must be moved to the package generated for the package in which the table is defined. These definitions, as well as all others now being discussed for the EMPLOYEE table, would be placed in package B in this example. There are still many generated definitions, not discussed here, that depend only on the user-defined data types and not on any specific tables within a database. These definitions are still placed in the package generated from the one in which the user type definitions appear, for the rationale already discussed.

Now the definitions of the "<=" functions must be added so that each will accept, as its left parameter, an object of the appropriate table- and type-specific column name data type. Also, the returned result should be typed according to the table being updated. Instead of returning a value of type SET_CLAUSE, it will return a value of a table-specific type that is derived from type SET_CLAUSE. It will be shown later how Ada's overloading resolution ties this all together to select the correct functions to implement our update statement. Using a similar naming convention as for the types derived from TABLE_NAME, suffixing the table name with _SET_CLAUSE, the table-specific type derived from SET_CLAUSE would be generated as:

```
type EMPLOYEE_SET_CLAUSE is new SET_CLAUSE;
```

And the "<=" functions generated are:

```
function "<=" ( LEFT  : EMPLOYEE_EMPLOYEE_NAME_COLUMN_NAME;
                RIGHT : EMPLOYEE_NAME ) return EMPLOYEE_SET_CLAUSE;
function "<=" ( LEFT  : EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME;
                RIGHT : EMPLOYEE_STATUS ) return EMPLOYEE_SET_CLAUSE;
```

The UPDATE procedure called by the update statement must now be generated to handle the table-specific types returned by the functions just defined.

```
procedure UPDATE ( TABLE : in EMPLOYEE_TABLE_NAME;
                   SET   : in EMPLOYEE_SET_CLAUSE;
                   WHERE : in SEARCH_CONDITION );
```

The corresponding definitions that would be generated for the EMPLOYEE_HISTORY table are shown below for completeness and comparison. These definitions would be placed in package C in the example, generated from package C_SCHEMA in which the EMPLOYEE_HISTORY table is defined.

```
type EMPLOYEE_HISTORY_TABLE_NAME is new TABLE_NAME;

function EMPLOYEE_HISTORY return EMPLOYEE_HISTORY_TABLE_NAME;

type EMPLOYEE_HISTORY_EMPLOYEE_NAME_COLUMN_NAME   is new COLUMN_NAME;
type EMPLOYEE_HISTORY_EMPLOYEE_STATUS_COLUMN_NAME is new COLUMN_NAME;

function NAME   return EMPLOYEE_HISTORY_EMPLOYEE_NAME_COLUMN_NAME;
function STATUS return EMPLOYEE_HISTORY_EMPLOYEE_STATUS_COLUMN_NAME;

type EMPLOYEE_HISTORY_SET_CLAUSE is new SET_CLAUSE;

function "<=" ( LEFT  : EMPLOYEE_HISTORY_EMPLOYEE_NAME_COLUMN_NAME;
                RIGHT : EMPLOYEE_NAME ) return
                                  EMPLOYEE_HISTORY_SET_CLAUSE;
function "<=" ( LEFT  : EMPLOYEE_HISTORY_EMPLOYEE_STATUS_COLUMN_NAME;
                RIGHT : EMPLOYEE_STATUS ) return
                                  EMPLOYEE_HISTORY_SET_CLAUSE;

procedure UPDATE ( TABLE : in EMPLOYEE_HISTORY_TABLE_NAME;
                   SET   : in EMPLOYEE_HISTORY_SET_CLAUSE;
                   WHERE : in SEARCH_CONDITION );
```

Note, for example, that the two STATUS functions are now not only distinctly defined, but also return values of distinct types. It will be shown how the Ada compiler uses these distinct types so that its overloading resolution selects the appropriate STATUS function in the example.

3.13. Strong Typing and Separate Compilation Support

In discussing how overloading resolution is applied to process the update statement, it is not implied that any actual Ada compiler would follow the identical line of reasoning. It is merely meant to demonstrate that it is possible to uniquely select all functions as required to process the statement. Any Ada compiler will arrive at the same result, by whatever route is programmed for it. Also, the discussion will be limited to those functions discussed here. There are, for example, several other EMPLOYEE functions defined for uses of the EMPLOYEE table name in other contexts. These functions must also be considered in the Ada compiler's overloading resolution. The discussion will be simplified, however, by only considering those functions already discussed.

How are the calls to the UPDATE procedure, such as this one, processed:

```
UPDATE ( EMPLOYEE,
  SET => STATUS <= RETIRED . . . );
```

Several UPDATE procedures have been defined, so it isn't known yet which one is being called! However, there is only one EMPLOYEE function (that has been discussed, actually there is only one returning a type that is acceptable for the first parameter to any of the UPDATE procedures), and that is:

```
function EMPLOYEE return EMPLOYEE_TABLE_NAME;
```

There is only one UPDATE procedure defined whose first parameter is of type EMPLOYEE_TABLE_NAME, and that is:

```
procedure UPDATE ( TABLE : in EMPLOYEE_TABLE_NAME;
                   SET   : in EMPLOYEE_SET_CLAUSE;
                   WHERE : in SEARCH_CONDITION );
```

Selecting this unique UPDATE procedure determines that the SET parameter must be of type EMPLOYEE_SET_CLAUSE. The value of the SET parameter in the update statement is the result returned by a call to the "<=" function. There are several "<=" functions returning a value of type EMPLOYEE_SET_CLAUSE. However, the right parameter (RETIRED) to the "<=" function is a program literal of type EMPLOYEE_STATUS. There is only one "<=" function defined to accept a right parameter of type EMPLOYEE_STATUS and to return a value of type EMPLOYEE_SET_CLAUSE, and that is:

```
function "<=" ( LEFT  : EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME;
                RIGHT : EMPLOYEE_STATUS ) return EMPLOYEE_SET_CLAUSE;
```

Selecting this unique "<=" function determines that its left parameter must be of type EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME. In the update statement, the value of the "<=" function's left parameter is that returned by the call to the STATUS function. And there is only one STATUS function defined that returns a value of type EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME, namely:

```
function STATUS return EMPLOYEE_EMPLOYEE_STATUS_COLUMN_NAME;
```

Thus, the functions that are called in processing the update statement have been completely determined. The added complexity of generating table-specific types permits the form of an Ada/SQL update statement to be as close as possible to the form of the corresponding SQL update statement. As mentioned before, where there was a design decision required because faithfulness to SQL could be achieved only at the expense of reasonable additional implementation complexity, the decision was made in favor of faithfulness to SQL. Application programmers would see any unnecessary deviations from SQL; they never see the complexities of implementation such as has been described. All the types and subprograms discussed are automatically generated from the DDL and are of no concern to application programmers, who need only know that the generated code magically enables their Ada/SQL programs to compile and execute.

3.14. Summary

It has been shown how types and subprograms are generated for only two tables, each with two columns of two different data types. Imagine how many types and subprograms must be generated for a typical database, with its many tables, columns, and user-defined data types! Also, the discussion has been limited to one small piece of a single Ada/SQL statement. Imagine again how many additional types and subprograms are required for the other syntactic classes represented within the Ada/SQL language! Quite a lot! How will this affect ultimate program size, and therefore practicality?

The type definitions do not, of course, require any program space at runtime; the information they provide is used at compile time. All the subprograms are generated by instantiating generic packages in which they are defined. The definitions of these generic packages are such that a smart Ada compiler could reuse much of the code from one instantiation in another. For most of the subprograms, each instantiation of their defining package would actually generate identical code for the same subprogram. This is because each version of such a subprogram will differ only in the types of its parameters and (for functions) returned result, where corresponding types in the various versions are all derived from the same parent type. (The UPDATE procedure shown above is an example of this.) The same single copy of compiled code can be used to implement the various versions of such subprograms: all type checking required by the different parameter and result types is performed at compile time and does not affect the executable code.

So, it is not expected that Ada/SQL development mode programs will be excessively large, once smart compilers are available. As already noted, application programmers do not have to be concerned with the complexities of all these generic packages, since the required code is automatically generated from the data definition language. And, once application debugging is complete and performance is important, a production mode Ada/SQL system can be used. As stated earlier, a production mode system preprocesses Ada/SQL statements in conjunction with the DBMS, to replace Ada/SQL DML statements with calls to database procedures that have already been processed for maximum database operation efficiency. None of the generated code described here is produced for a production mode Ada/SQL system.

Ada/SQL exhibits many features that are important in an Ada database interface. First, application programs that use

Ada/SQL can be run on any computer system using any underlying DBMS. This database independence is achieved by placing database dependent details within the interface implementation, leaving the application program completely portable. In addition, the interface is written using per Ada, in line with the philosophy of not subsetting or extending the language. Ada/SQL has been designed such that the resulting code satisfies the readability, reliability, and maintainability objectives of Ada itself.

4. References

[ADA 83]
Military Standard: Ada Programming Language, ANSI/MIL-STD-1815A, 1983.

[ANSI 85]
Draft Proposed American National Standard: Database Language SQL. New York, NY: American National Standards Institute, Inc., 1985.

[DATE 83]
C.J. Date, "Database: A Primer," Reading, MA: Addison-Wesley, 1983.

Appendix I
Ada/SQL Binding Specification

I.1. Introduction

This document defines the Ada procedure language access to the draft proposed American National Standard (dpANS) "Database Language SQL", ISO document TC97/SC21/WG3-N96 and ANSI document X3H2-86-2, dated January 1986. A native language approach is used in defining embedded Ada SQL programs, such that standard, validated Ada compilers may be used to translate Ada programs containing embedded SQL. This is accomplished by slightly adjusting SQL syntax so that the resulting "Ada/SQL" conforms to Ada syntax. SQL operations, as well as database names, are subprogram calls in Ada/SQL. This document defines how Ada/SQL syntax differs from the dpANS syntax.

As used herein, an "implementation" consists of two parts: (1) a database management system (DBMS) providing the functions required for SQL access to data, and (2) an interface enabling Ada programs to access data controlled by the DBMS.

I.2. INTRODUCTORY DIFFERENCES

The introductory section of the dpANS applies to this Standard as well, except as noted below (section numbers are from the dpANS):

1.7(2) - An implementation conforming to this Standard must implement embedded SQL Ada ("<embedded SQL Ada program>") and the Ada schema definition language ("<schema>").

1.7(3) - A MIL-STD implementation conforming to this Standard must conform to Level 2 of the dpANS. An ANSI standard implementation may conform to either Level 1 or Level 2.

1.7(5) - An implementation conforming to this Standard may not allow embedded SQL Ada options not specified by this Standard or the dpANS.

I.3. CONCEPTS

The concepts section of the dpANS applies to this Standard as well, except as noted below (section numbers are from the dpANS):

2.2(4) - A non-null value may be of any Ada data type, other than access or task, i.e., the value of any non-access, non-task program object may be stored within a database column. For the purpose of this definition, any composite type containing an access or task type component shall also be considered an access or task type. Types which use pointers to refer to external objects, such as files, shall likewise be considered to be access types.

2.2.2 - This section shall apply to all Ada scalar types -- enumeration, integer, floating point, and fixed point. The dpANS concept of "exact numeric value" applies to Ada integer types, while the dpANS concept of "approximate numeric value" corresponds to both Ada floating and fixed point types. The interface must cause enumeration types to be stored by the DBMS such that DBMS and Ada comparison operators return consistent results. Assignment is only permitted between comparable objects. The Ada/SQL concept of comparability is defined in section 3.9.

2.2.3 - This is a new section for Ada record and array types. Record and array objects of the same type are comparable. SQL operations on array and record types are described in sections 4.7(3.6) and 4.7(3.7) of this manual.

2.3(2) - See Ada data definition language (section 4) for column descriptions.

2.5(2) - The UNIQUE_ERROR exception will be raised following an operation violating a UNIQUE constraint, and the NULL_ERROR exception will be raised following an operation violating a NOT NULL constraint. An implementation

may select which exception to raise if more than one error occurs within a single operation: programs relying on any particular exception in this case are erroneous.

2.10.1 Status in Ada/SQL is returned by raising the appropriate exception on error. The following exceptions are defined, based on similar error conditions defined in the dpANS:

```
UNIQUE_ERROR     : exception;
NULL_ERROR       : exception;
NOT_FOUND_ERROR  : exception;
```

Additional exceptions will be defined in a later version of this standard.

2.11 - This Standard specifies the actions of Ada/SQL statements embedded within legal Ada programs. A "legal Ada program" meets the conformance criteria of the Ada Programming Language Military Standard (ANSI/MIL-STD-1815A), hereafter referred to as the Language Reference Manual (LRM).

2.12(2) - The <module> concept is not relevant to Ada/SQL, since SQL is embedded within Ada programs rather than contained within separate <module>s. A cursor is created by execution of a <declare cursor>, and not destroyed until the program defining it terminates.

2.14(1) - An <embedded SQL Ada program> uses exact Ada syntax that may be compiled by any standard, validated Ada compiler. The embedded SQL syntax is adjusted to conform to Ada syntax.

2.15(4) - A <schema> has a single <authorization identifier>. It may, however, be defined within one or more <schema package declaration>s. Each schema package declaration is an Ada package. The SQL syntax (see 3.4 for exceptions)

```
<table name> ::= <authorization identifier> . <table identifier>
```

may be used to select a specific table, but the Ada syntax of

```
<package name> . <subprogram name>
```

may also be used, where <package name> selects the <schema package declaration> and <subprogram name> selects the table within that package.

2.15(5) - The applicable <schema package declaration> and corresponding SQL <authorization identifier> are selected according to Ada visibility rules for table names without an explicitly stated authorization identifier or package name prefix.

2.16 - Valid execution of any Ada/SQL data manipulation statement other than <declare cursor> initiates a transaction for the executing program, if one is not already in progress. A transaction in progress upon program termination is automatically terminated as if a <rollback statement> had been issued.

I.4. COMMON ELEMENTS

The first five common elements of the dpANS apply to this Standard as well, except as noted below (section numbers are from the dpANS).

3.2 - <literal>s, since they are compiled by an Ada compiler, must be specified with Ada syntax (syntactic elements other than <literal> as in LRM):

```
<literal> ::= numeric_literal      -- integer, floating and fixed point
            | enumeration_literal  -- enumeration
            | string_literal       -- array of character (STRING)
            | aggregate            -- record, array
```

3.3 - Lexical units in Ada/SQL are as in Ada. Ada reserved words can obviously not be used as identifiers (SQL database names), and Ada/SQL key words should also not be used as program and database variables, to avoid confusion. There are, however, no specific restrictions beyond those imposed by Ada.

3.4.SR(1) - In most contexts where an <authorization identifier> is used within a <table name>, the <authorization identifier> is separated from the <table identifier> by a period. There are, however, isolated occurrences where the separator character is an underscore (see section 3.20) or a hyphen (see sections 4.5a and 6.7).

3.4.SR(2) - Ada visibility rules determine the <authorization identifier> of an unqualified <table name>.

3.4.SR(7) - <correlation name>s must be explicitly declared to pertain to specific tables, as described in section 3.20. The same <correlation name> may be reused within different scopes of the same statement, although it must refer to (different instances of) the same table.

3.4.SR(9) - <module name> is not used for embedded language.

3.4.SR(11) - <procedure name> is not used for embedded language.

3.4.SR(12) - <parameter name> is not used for embedded language.

3.5 Any Ada data type may be declared within a schema, except for access and task types, and composite types containing access or task subcomponents. Type declarations follow standard Ada syntax. Expressions used in type declarations must be static or record discriminants. Additional description of data types is provided in Section 4.

I.5. REMAINING SYNTACTIC/SEMANTIC DIFFERENCES

The Ada/SQL equivalents of the remaining dpANS syntactic/semantic sections are now given. The following aspects are discussed for each section:

FUNCTION - A concise description of the function of language element discussed

EXAMPLE - Examples of use within Ada/SQL programs. The data manipulation examples use the following table:

```
type ANALYST is
  record
    NAME    : ANALYST_NAME_NOT_NULL_UNIQUE;
    SALARY  : ANALYST_SALARY;
    MANAGER : ANALYST_NAME;
  end record;
```

FORMAT - BNF and commentary description of the syntactic use of the language element within Ada/SQL programs

3.6 <value specification>

FUNCTION:

(1) Indicate values of embedded variables, (2) indicate whether or not the values are null, (3) implement the keyword USER.

EXAMPLE:

```
NEW_EMPLOYEE_NAME       : ANALYST_NAME;
NEW_EMPLOYEE_SALARY     : ANALYST_SALARY;
SALARY_KNOWN            : INDICATOR_VARIABLE;
CURRENT_MANAGER         : MANAGER_NAME;
CURSOR                  : CURSOR_NAME; -- see section 6.1
. . .
INSERT_INTO ( ANALYST ( NAME & SALARY & MANAGER ),
      VALUES <= NEW_EMPLOYEE_NAME                              -- 1
            and INDICATOR(NEW_EMPLOYEE_SALARY, SALARY_KNOWN)   -- 2
            and USER );                                        -- 3

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => EQ ( MANAGER , INDICATOR(CURRENT_MANAGER) ) -- 4
  AND        SALARY > 25_000.00 ) );                   -- 5
```

FORMAT:

```
<value specification> ::=
     <variable specification>
   | <literal>
   | USER
```

<parameter specification>s are not used within Ada/SQL since an embedded language, rather than a module language, is supported. The keyword USER may be specified; it is an Ada function (see example 3).

```
<variable specification> ::=
      <Ada program expression>
    | INDICATOR ( <Ada program expression> [ , <indicator variable> ] )

<Ada program expression> ::= program expression of type appropriate for
     data base column being accessed

<indicator variable> ::= program variable of type  INDICATOR_VARIABLE,
     which is an enumeration type with values NULL_VALUE and NOT_NULL
```

Unlike SQL <embedded variable name>s, program variables within Ada/SQL expressions are not preceded with colons (see example 1). Also, general program expressions may be used as <variable specification>s in Ada/SQL; SQL permits only host variable names. Where an <indicator variable> is desired, it is necessary to have a function in order to combine both the value and the indicator into a single syntactic element. This function is called INDICATOR (see example 2). For convenience in certain contexts, the <indicator variable> may be omitted from the call to INDICATOR, and defaults to NOT_NULL (see example 4, where CURRENT_MANAGER could also have been used by itself, without the surrounding call to INDICATOR).

<literal> ::= see section 3.2

Ada literals are program expressions, so no special syntax is required (see example 5).

3.7 <column specification>

FUNCTION:

Indicate values of database columns.

EXAMPLE:

```
package E is new ANALYST_CORRELATION_NAME; -- employees \ see section
package M is new ANALYST_CORRELATION_NAME; -- managers  / 3.20

CURSOR : CURSOR_NAME; -- see section 6.1
. . .


DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => ANALYST,
  WHERE  => SALARY > 25_000.00 ) );                       -- 1

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM   => ANALYST,
  WHERE  => ANALYST.SALARY > 25_000.00 ) );               -- 2

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( E.NAME & E.SALARY & M.NAME & M.SALARY,          -- 3
  FROM   => E.ANALYST & M.ANALYST, -- see section 3.20
  WHERE  => EQ ( E.MANAGER , M.NAME )                     -- 3
  AND         E.SALARY > M.SALARY ) );                    -- 3
```

FORMAT:

No syntax changes required. Example 1 shows a <column specification> without a <qualifier>, example 2 shows a <table name> used as a <qualifier>, and example 3 shows <correlation name>s used as <qualifier>s.

3.8 <set function specification>

FUNCTION:

Compute aggregate functions on database values.

EXAMPLE:

```
NUMBER  : DATABASE.INT; -- see 4.7(3.5.4)
AVERAGE : ANALYST_SALARY;

. . .
SELEC ( COUNT('*'),                -- 1
FROM => ANALYST );
INTO(NUMBER);

SELEC ( COUNT_DISTINCT(MANAGER),   -- 2
FROM => ANALYST );
INTO(NUMBER);

SELEC ( AVG(SALARY),               -- 3
FROM => ANALYST );
INTO(AVERAGE);

SELEC ( AVG_ALL(SALARY),           -- 4
FROM => ANALYST );
INTO(AVERAGE);
```

FORMAT:

```
<set function specification> ::=
     COUNT ( '*' )  | <distinct set function> | <all set function>
```

An asterisk by itself cannot be used as an argument to an Ada function, so Ada/SQL encloses it in quotes to make it a character literal (see example 1).

```
<distinct set function> ::=
     { AVG_DISTINCT | MAX | DISTINCT | MIN_DISTINCT | SUM_DISTINCT |
          COUNT_DISTINCT } ( <column specification> )

<column specification> ::= see section 3.7
```

The DISTINCT cannot stand by itself, so it is included in the function name (see example 2).

```
<all set function> ::=
     { AVG | MAX | MIN | SUM |  AVG_ALL | MAX_ALL | MIN_ALL | SUM_ALL }
        ( <value expression> )

<value expression> ::= see section 3.9
```

The ALL can likewise not stand by itself, and is brought into the function name (see examples 3 and 4).

The value returned by a set function, other than a count set function, is typed the same as the <column specification> or <value expression> argument of the set function. The value returned by a count set function is of type INT defined in the DATABASE package, as described in section 4.7(3.5.4).

3.9 <value expression>

FUNCTION:

Specify a (possibly) computed value.

EXAMPLE:

```
NUMBER : DATABASE.INT; -- see section 4.7(3.5.4)
. . .
SELEC  ( COUNT('*') ,
FROM  => ANALYST ,
WHERE => ( SALARY + 1000.00 ) / 2080.0 < + 3.85 );
INTO(NUMBER);
```

FORMAT:

No syntax changes required: Ada/SQL supports all SQL operators with virtually the same precedences. The only difference is that SQL permits monadic "+" or "-" operators before any <primary> used within a <value expression>. The corresponding Ada unary_adding_operators may be applied only to an entire simple_expression. Furthermore, a leading Ada unary_adding_operator is applied to the entire first term within a simple_expression, while a leading SQL monadic operator in a similar <value expression> would be applied to the first <factor> within the <term>. Expressions written in Ada/SQL are interpreted according to Ada rules. Due to the nature of the operations, however, the arithmetic results will be the same as if SQL interpretation had been applied. Furthermore, any SQL <value expression> may be equivalently stated in Ada, using parentheses or depending on the properties of the arithmetic operators, even though the Ada syntax is more restrictive.

Arithmetic operators may be applied only to scalar numeric types. Both arguments must be of comparable types. In Ada/SQL, two database columns are comparable if they each contain values of the same Ada type, and a database column and a program value are comparable if the Ada program value is of the same type as the values contained within the database column.

3.10 <predicate>

FUNCTION:

Specify a condition that can be evaluated to give a truth value of "true", "false", or "unknown".

EXAMPLE:

See discussions on individual predicate types.

FORMAT:

No syntax changes required; all types of predicate are supported by Ada/SQL.

3.11 <comparison predicate>

FUNCTION:

Specify a comparison of two values.

EXAMPLE:

```
package E is new ANALYST_CORRELATION_NAME; -- see section 3.20

CURSOR : CURSOR_NAME; -- see section 6.1
. . .

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => E.ANALYST, -- see section 3.20
  WHERE => SALARY >                       -- 1
    SELEC  ( AVG(SALARY),
    FROM  => ANALYST,
    WHERE => EQ(MANAGER,E.MANAGER)  )  )  ); -- 2
```

FORMAT:

```
<comparison predicate> ::=
     <equality operator> ( <value expression> , <right comparison
     operand> )
   | <value expression> <ordering operator> <right comparison operand>
```

Although Ada supports all the SQL comparison operators, restrictions on overloading = and /= prevent them from being used in Ada/SQL. Instead, functions EQ and NE are defined for these <equality operator>s. The other <ordering operator>s are expressed in their natural notation. Example 2 shows an <equality operator> function; example 1 shows an <ordering operator>. <equality operators> are available for all user-defined types. <ordering operators> are available for all scalar types and for character arrays with a single integer index (represented in SQL as strings).

```
<equality operator> ::=
     EQ | NE

<ordering operator> ::=
     < | >  |  <= | >=

<value expression> ::= see section 3.9

<right comparison operand> ::=
     <value expression> | <sub-query>
```

The right operand of a comparison predicate may be either a (possibly computed) value (see example 2) or a <sub-query> (see example 1).

<sub-query> ::= see section 3.24

3.12 <between predicate>

FUNCTION:

Specify a range comparison.

EXAMPLE:

```
CURSOR : CURSOR_NAME; -- see section 6.1
. . .
DECLAR  (  CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => BETWEEN(SALARY,20_000.00 AND 30_000.00) ) );

           -- variations: NOT BETWEEN
```

FORMAT:

```
<between predicate> ::=
     [ NOT ] BETWEEN ( <value expression> ,
                          <value expression> AND <value expression> )
```

BETWEEN cannot be written as an infix operator in Ada; it is instead made a function of two parameters. The first parameter is the value to be tested, the second parameter is the range, with the keyword AND joining the endpoint <value expression>s. No special function is required to implement the NOT option, the same overloaded NOT operator used with <search condition>s can be used to negate a BETWEEN test. The BETWEEN function is only defined for types on which the <ordering operator>s are defined (see section 3.11).

<value expression> ::= see section 3.9

3.13 <in predicate>

FUNCTION:

Specify a quantified comparison.

EXAMPLE:

```
PRIMARY_MANAGER,
ALTERNATE_MANAGER : MANAGER_NAME;
CURSOR            : CURSOR_NAME; -- see section 6.1
. . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => IS_IN ( MANAGER , PRIMARY_MANAGER or ALTERNATE_MANAGER ) ));

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE NOT_IN ( MANAGER ,
    SELEC      ( MANAGER,
    FROM     => ANALYST,
    GROUP_BY => MANAGER,
    HAVING   => AVG(SAL) > 20_000.00 ) ) ) );
```

FORMAT:

```
<in predicate> ::=
   { IS_IN | NOT_IN }
      ( <value expression> , { <sub-query> | <in value list> } )
```

The Ada "in" operator cannot be overloaded, so IS_IN is used instead of the SQL IN, and NOT IN becomes NOT_IN. These new operators then cannot be expressed in infix notation, so they become functions of two parameters. The first parameter is the <value expression> to be tested for set membership or non-membership, and the second parameter is the specification of the set to be tested. Parentheses are not required around the <in value list> because (1) the number of closing parentheses would get cumbersome, since the closing parenthesis for the function must follow the <in value list> anyway, and (2) they are not required by Ada syntax, so the compiler cannot check whether they are used or not.

```
<value expression> ::= see section 3.9

<sub-query> ::= see section 3.24

<in value list> ::=
  <value specification> [ { or <value specification> } ... ]
```

Items in a value list cannot be separated by commas, so "or" is used instead. This corresponds to the semantics that a record is selected if its <value expression> equals the first <value specification> OR the second one, etc.

<value specification> ::= see section 3.6

3.14 <like predicate>

FUNCTION:

Specify a pattern-match comparison.

EXAMPLE:

```
LAST_NAME : ANALYST_NAME;
CURSOR    : CURSOR_NAME; -- see section 6.1
. . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => LIKE ( NAME , "%" & LAST_NAME ) ) ); -- variation: NOT LIKE
. . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => LIKE ( NAME , "%" & LAST_NAME , ESCAPE => "^" ) ) );
```

FORMAT:

```
<like predicate> ::=
     [ NOT ] LIKE ( <column specification> , <pattern>
                  [ , ESCAPE => <escape character> ] )
```

LIKE cannot be written as an infix operator in Ada; it is instead made a function of three parameters. The first parameter is the specification of the column to be tested, the second parameter is an array of characters containing the appropriate pattern matching characters. No special function is required to implement the NOT option, the same overloaded NOT operator used with <search condition>s can be used to negate a LIKE test. LIKE is only defined for arrays of characters with a single integer index (represented in SQL as strings), with all parameters of the same type. To be useful, the component type must include the pattern matching characters, which are '_' and '%'.

The third parameter to LIKE is named ESCAPE, to implement the SQL keyword for the escape character. This parameter is optional, and defaults to no escape character specified. If an escape character is specified, it must be an array of length 1, of the same type as the pattern.

<column specification> ::= see section 3.7

<pattern> ::= <value specification>

<escape character> ::= <value specification>

<value specification> ::= see section 3.6

3.15 <null predicate>

FUNCTION:

Specify a test for a null value.

EXAMPLE:

```
CURSOR : CURSOR_NAME; -- see section 6.1
. . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => IS_NULL(MANAGER) ) ); -- variation: IS_NOT_NULL
```

FORMAT:

```
<null predicate> ::=
   { IS_NULL | IS_NOT_NULL } ( <column specification> )
```

IS NULL and IS NOT NULL cannot be written as postfix operators in Ada, they are instead made functions IS_NULL and IS_NOT_NULL. The functions take as their parameter the specification of the column to be tested.

<column specification> ::= see section 3.7

3.16 <quantified predicate>

FUNCTION:

Specify a quantified comparison.

EXAMPLE:

```
package E is new ANALYST_CORRELATION_NAME; -- see section 3.20

CURSOR : CURSOR_NAME; -- see section 6.1
. . .

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => E.ANALYST, -- see section 3.20
  WHERE => SALARY >= ALLL (       -- 1
    SELEC  ( SALARY,
    FROM  => ANALYST,
    WHERE => EQ(MANAGER,E.MANAGER  )  )  )  )  ) ;

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => EQ ( NAME , ANY (       -- 2      variation: SOME
    SELEC  ( MANAGER,
    FROM  => ANALYST  )  )  )  )  ) ;
```

FORMAT:

```
<quantified predicate> ::=
     <equality operator> ( <value expression> , <quantified sub-query> )
   |  <value expression> <ordering operator> <quantified sub-query>
```

The syntax and considerations for <quantified predicate>s are the same as for <comparison predicate>s (see section 3.11), where <quantified sub-query>s are used instead of <sub-queries>.

<equality operator> ::= see section 3.11

<ordering operator> ::= see section 3.11

<value expression> ::= see section 3.9

```
<quantified sub-query> ::=
     <quantifier> ( <sub-query> )
```

The <quantifier> cannot stand by itself in Ada; it is made into a function with the <sub-query> being its parameter.

```
<quantifier> ::=
     <all> | <some>

<all> ::= ALLL
```

ALL is an Ada reserved word. ALLL is used instead.

```
<some> ::= SOME   |  ANY

<sub-query> ::= see section 3.24
```

3.17 <exists predicate>

FUNCTION:

Specify a test for an empty set.

EXAMPLE:

```
package E is new ANALYST_CORRELATION_NAME; -- see section 3.20

CURSOR : CURSOR_NAME; -- see section 6.1
. . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => E.ANALYST, -- see section 3.20
  WHERE => EXISTS (
    SELEC  ( '*',
    FROM  => ANALYST,
    WHERE => EQ(MANAGER,E.NAME) ) ) ) );
```

FORMAT:

```
<exists predicate> ::=
     EXISTS ( <sub-query> )
```

EXISTS is an Ada function, with the <sub-query> being its parameter.

<sub-query> ::= see section 3.24

3.18 <search condition>

FUNCTION:

Specify a condition that is "true", "false", or "unknown" depending on the result of applying boolean operators to specified conditions.

EXAMPLE:

```
PRIMARY_MANAGER,
ALTERNATE_MANAGER : MANAGER_NAME;
CURSOR            : CURSOR_NAME; -- see section 6.1
  . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC  ( '*',
  FROM  => ANALYST,
  WHERE => NOT BETWEEN ( SALARY , 20_000.00 AND 30_000.00 )
  AND     ( EQ(MANAGER,PRIMARY_MANAGER)
  OR        EQ(MANAGER,ALTERNATE_MANAGER) ) ) );
```

FORMAT:

```
<search condition> ::=
       <boolean factor> [ { AND <boolean factor> } ... ]
     |  <boolean factor [ { OR  <boolean factor> } ... ]

<boolean factor> ::=
     [ NOT ] <boolean primary>

<boolean primary> ::=
     <predicate> | ( <search condition> )

<predicate> ::= see section 3.10
```

The Ada AND, OR, and NOT operators correspond to those of SQL. However, Ada requires that combinations of ANDs and ORs be parenthesized to clearly show order of evaluation, whereas SQL provides precedence of AND over OR. The extra parentheses required by Ada are permitted by SQL, so a legal Ada <search condition> will still correspond to a valid SQL one, and will be interpreted in a consistent fashion.

3.19 <table expression>

FUNCTION:

Specify a table or a grouped table.

EXAMPLE:

Examples in other sections include use of <table expression>s.

FORMAT:

```
<table expression> ::=
     <from clause>
   [ , <where clause> ]
   [ , <group by clause> ]
   [ , <having clause> ]
```

The various SQL clauses are parameters to Ada/SQL subprograms. Each clause is therefore separated from the one before it with a comma. The syntax for the <from clause> does not show the preceding comma because <table expression> is preceded with a comma wherever it is used in the grammar.

```
<from clause> ::= see section 3.20

<where clause> ::= see section 3.21

<group by clause> ::= see section 3.22

<having clause> ::= see section 3.23
```

3.20 <from clause>

FUNCTION:

Specify a table derived from one or more named tables.

EXAMPLE:

Examples in other sections include use of <from clause>s.

FORMAT:

```
<from clause> ::=
     FROM => <table reference> [ { & <table reference> } ... ]
```

The <table reference>s cannot be separated from each other by commas in Ada, so ampersands are used instead. Since the <from clause> is actually a parameter to a function, the named parameter association symbol => follows the keyword FROM.

```
<table reference> ::=
     [ <correlation name> . ] <table name>
```

In Ada/SQL. <correlation name>s are actually Ada packages. The <correlation name>, if used, must therefore precede the <table name>, with the <table name> being a function selected from the <correlation name> package. Appropriate functions are also generated so that the <table name> may be referenced without the optional <correlation name>.

```
<table name> ::= the name of a database table, defined as an overloaded
    function within the database-specific portion of the underlying
    Ada/SQL definitions

<correlation name> ::= a package instantiated from the generic package
     specific to each <table name>.  The generic packages are produced by the
     SQL function generator.  In order to define a <correlation name>, the
     appropriate generic package must be instantiated, in one of the two
     following ways:

          package <correlation name> is new
            <table identifier>_CORRELATION_NAME;

          package <correlation name> is new
            <authorization identifier>_<table identifier>_CORRELATION_NAME;
```

Note that the generic packages are, in general, named <table name>_CORRELATION_NAME, except that an <authorization identifier> used within a <table name> is separated from the <table identifier> by an underscore instead of a period as in SQL. This is an exception; Ada/SQL syntax for <table name>s is identical to that of SQL, except where otherwise noted.

Although <correlation name>s are specifically declared to pertain to specific tables, the same <correlation name> may be reused within different scopes of the same statement, to refer to different instances of the same table.

See section 3.7 for an example of using <correlation name>s.

3.21 <where clause>

FUNCTION:

Specify a table derived by the application of a <search condition> to the result of the preceding <from clause>.

EXAMPLE:

Examples in other sections include use of <where clause>s.

FORMAT:

```
<where clause> ::=
     WHERE => <search condition>
```

Since the <where clause> is actually a parameter to a function, the named parameter association symbol => follows the keyword WHERE.

```
<search condition> ::= see section 3.18
```

3.22 <group by clause>

FUNCTION:

Specify a grouped table derived by the application of the <group by clause> to the result of the previously specified clause.

EXAMPLE:

See section 3.13 for an example using a <group by clause>

FORMAT:

```
<group by clause> ::=
     GROUP_BY => <column specification> [ { & <column specification> } ... ]
```

Since the <group by clause> is actually a parameter to a function, the named parameter association symbol => follows the keyword GROUP_BY, which requires the underscore to make it a single lexical symbol to Ada. <column specification>s cannot be separated by commas: ampersands are used instead.

```
<column specification> ::= see section 3.7
```

3.23 <having clause>

FUNCTION:

Specify a grouped table derived by the application of the <having clause> to the result of the previously specified clause.

EXAMPLE:

See section 3.13 for an example using a <having clause>

FORMAT:

```
<having clause> ::=
     HAVING => <search condition>
```

Since the <having clause> is actually a parameter to a function, the named parameter association symbol => follows the keyword HAVING.

```
<search condition> ::= see section 3.18
```

3.24 <sub-query>

FUNCTION:

Specify a multi-set of values derived from the result of a <table expression>.

EXAMPLE:

Examples in other sections include use of <sub-query>s.

FORMAT:

```
<sub-query> ::=
   [ SELEC | SELECT_ALL | SELECT_DISTINCT ]
       ( <sub-query result specification> , <table expression> )
```

SELECT used as a <sub-query> is an Ada function. It is not possible to specify the ALL or DISTINCT keywords separately, so they are part of the function name if used. The name of the function is SELEC if neither keyword is used, since SELECT is an Ada reserved word. The <sub-query> functions have five parameters which must, of course, be surrounded by parentheses and separated by commas. Any or all of the last three parameters may be omitted, since named associations are used for them and they have default values indicating their omission. The first parameter is the <sub-query result specification>, while the second through fifth parameters are the FROM, WHERE, GROUP BY, and HAVING clauses from the <table expression>, and so are named FROM, WHERE, GROUP_BY, and HAVING, respectively. Parentheses are not required around the <sub-query> because (1) the number of closing parentheses would get cumbersome, since the function call itself also provides a closing parenthesis, (2) a <sub-query> is often an argument to a function, which causes it to be surrounded by parentheses anyway, and (3) parentheses are not required by Ada syntax, so the compiler cannot check whether they are used or not.

```
<sub-query result specification> ::=
      <value expression>
    | '*'
```

An asterisk cannot stand alone by itself as an argument to an Ada function, so it is enclosed in quotes to make it a character literal.

```
<value expression> ::= see section 3.9

<table expression> ::= see section 3.19
```

3.25 <query specification>

FUNCTION:

Specify a table derived from the result of a <table expression>.

EXAMPLE:

Examples in other sections include use of <query specification>s.

FORMAT:

```
<query specification> ::=
   [ SELEC | SELECT_ALL | SELECT_DISTINCT ]
       ( <select list> , <table expression> )
```

The syntax and interpretation of a <query specification> is the same as for a <sub-query>, except that a <sub-query> retrieves only one column of values, while a <query specification> may retrieve more than one column.

```
<select list> ::=
        <value expression> [ { & <value expression> } ... ]
     |  '*'
```

The <value expression>s cannot be separated by commas, so ampersands are used instead. <value expression>s containing Ada binary_adding_operators may have to be enclosed in parentheses to enforce the correct precedence of their operators over the Ada/SQL ampersand connectives. An asterisk cannot stand alone by itself as an argument to an Ada function, so it is enclosed in quotes to make it a character literal.

```
<value expression> ::= see section 3.9

<table expression> ::= see section 3.19
```

4.1 <schema>

FUNCTION:

Ada/SQL schemas perform three functions: (1) Provide the Ada type definitions necessary for programs to declare variables to hold database values, (2) form input to a schema translator that converts Ada/SQL schemas into SQL schemas for creating database structures, and (3) form input to a SQL function generator that produces the functions and type definitions necessary to use schema database names within the Ada/SQL data manipulation language. The interrelation and automation of these functions provides program consistency checking within Ada/SQL. Ada/SQL schemas also contain sufficient information to be used as input to a test data generator to produce test data for populating databases defined by them.

EXAMPLE:

```
-- This package defines the ADMINISTRATION authorization identifier.
-- Authorization packages are used for two purposes:
-- (1) The authorization function defined (ADMINISTRATION in this example) is
--      referenced from a schema package to indicate the schema authorization
--      identifier, e.g., a schema package including the definition
--        SCHEMA_AUTHORIZATION : IDENTIFIER := ADMINISTRATION;
--      where ADMINISTRATION is the function defined herein, will be
--      considered part of the ADMINISTRATION schema.
-- (2) Other schemas reference the authorization function to grant privileges
--      to that authorization identifier.  The ADMINISTRATION function is
--      called in the body of COMPANYDB_TABLES_SCHEMA (below) for this
--      purpose.
-- Note how the ADMINISTRATION authorization package is independent of any
--  ADMINISTRATION schema that might be defined.  This is done by design to
--  minimize the number of recompilations required by changes to schemas.
--  Suppose, for example, that there were an ADMINISTRATION schema.  If the
--  schema is changed, must any other schemas granting privileges to
--  ADMINISTRATION be recompiled?  No, because they "with" only the
--  ADMINISTRATION authorization package, not any of the schema packages.
--  (Packages referencing tables within the ADMINISTRATION schema may, of
--  course, require recompilation.)
--  AUTHORIZATION_IDENTIFIER is a generic function defined within
--  SCHEMA_DEFINITION.  IDENTIFIER is a type defined within SCHEMA_-
--  DEFINITION.
--  The SCHEMA_DEFINITION package also includes the functions and procedures
--  necessary to define views and grant privileges.

with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION;

package ADMINISTRATION_AUTHORIZATION is

  function ADMINISTRATION is new AUTHORIZATION_IDENTIFIER;

end ADMINISTRATION_AUTHORIZATION;
-----------------------------------------------------------------------------
-- This is the authorization package for the PERSONNEL authorization
-- identifier.

with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION;

package PERSONNEL_AUTHORIZATION is
```

```
  function PERSONNEL is new AUTHORIZATION_IDENTIFIER;

end PERSONNEL_AUTHORIZATION;
-----------------------------------------------------------------------------
-- This is the authorization package for the COMPANYDB authorization
-- identifier.
-- The COMPANYDB schema is the only schema shown in this example.

with SCHEMA_DEFINITION;
 use SCHEMA_DEFINITION;

package COMPANYDB_AUTHORIZATION is

  function COMPANYDB is new AUTHORIZATION_IDENTIFIER;

end COMPANYDB_AUTHORIZATION;
-----------------------------------------------------------------------------
-- This package defines the data types used within the COMPANYDB schema.  As
-- can be seen, data types need not be defined within actual schema packages.
-- The ability to "with" definitions from other packages permits all the Ada
-- flexibilities of program organization.  Also, it is logical to define types
-- separately from schemas in those instances where some programs handle data
-- of those types without accessing a database. Such programs may then "with"
-- only the type definitions, and not the database definitions.

package COMPANYDB_TYPES is

  type EMPLOYEE_NAME is new STRING(1..15);
  subtype EMPLOYEE_NAME_NOT_NULL_UNIQUE is EMPLOYEE_NAME;

  type EMPLOYEE_AGE is range 14..100;
    -- many other type definitions would also use SQL NUMERIC(3,0)

  type EMPLOYEE_SAL is delta 0.01 range 0.00..999_999.99;
    -- many other type definitions would also use SQL DECIMAL(8,2)

  type DEPT_NAME is new STRING(1..10);
  subtype DEPT_NAME_NOT_NULL_UNIQUE is DEPT_NAME;

  type DEPT_LOC is new STRING(1..2);
  -- an enumeration type might also be used

end COMPANYDB_TYPES;
-----------------------------------------------------------------------------
-- This is the classification package for the COMPANYDB_TABLES schema package.
-- For the MIL-STD, every schema package must have a corresponding
-- classification package.  This is not required for the ANSI standard.  A
-- classification package defines the security classification of all columns
-- in each table defined by the corresponding schema package.  The
-- CLASSIFICATION_DEFINITION package defines a type CLASSIFICATION, which may
-- be adjusted to suit specific environments.
-- Although not specified for our simple example, it might be an enumeration
-- type such as type CLASSIFICATION is (UNCLASSIFIED,CONFIDENTIAL, SECRET,
-- TOP_SECRET);
-- In more complex environments, CLASSIFICATION might be a record type, with
-- components indicating releasability, special handling, and sensitive source
-- caveats as well as the standard four levels shown above. SECURITY_CLASSI-
```

```
-- FICTION is a generic function defined in CLASSIFICATION_DEFINITION, which is
-- instantiated for the most restrictive classification in the package being
-- declared.  The instantiated function (COMPANYDB_TABLES in this case) is
-- called from a schema package to indicate which classification package
-- applies to it.
-- Thus, COMPANYDB_TABLES_SCHEMA contains the declaration SECURITY :
-- CLASSIFICATION := COMPANYDB_TABLES; By design, the instantiated function
-- returns the classification with which it was instantiated, so that a program
-- referencing COMPANYDB_TABLES_SCHEMA.SECURITY can determine the most
-- restrictive classification applying to the data.  A classification package
-- defines record types paralleling those defining tables in the corresponding
-- schema -- the records have the same structure and component names, but
-- components are of type CLASSIFICATION in a classification package.  Default
-- values for the components are used to indicate classifications of the
-- columns.  The classification record types are all limited private to
-- indicate that classifications may not be arbitrarily adjusted by
-- application programs.  It is by design that classification records parallel
-- database records.  A system could store an associated classification record
-- with each data record stored, thereby marking each data value with a
-- classification. This is not required by the present standard, which assumes
-- that marking is at the column level; future standards may include syntax
-- for marking individual data values by setting classification records.

with CLASSIFICATION_DEFINITION;
 use CLASSIFICATION_DEFINITION;

package COMPANYDB_TABLES_CLASSIFICATION is

 function COMPANYDB_TABLES is new SECURITY_CLASSIFICATION
        (UNCLASSIFIED);

  type EMPLOYEE is limited private;

  type DEPT is limited private;

private

  type EMPLOYEE is
    record
      NAME : CLASSIFICATION := UNCLASSIFIED;
      AGE  : CLASSIFICATION := UNCLASSIFIED;
      SAL  : CLASSIFICATION := UNCLASSIFIED;
      DEPT : CLASSIFICATION := UNCLASSIFIED;
    end record;

  type DEPT is
    record
      NAME : CLASSIFICATION := UNCLASSIFIED;
      LOC  : CLASSIFICATION := UNCLASSIFIED;
    end record;

end COMPANYDB_TABLES_CLASSIFICATION;
-----------------------------------------------------------------------------
-- This is one of the two schema packages comprising the COMPANYDB schema.
-- The two packages together define the single SQL schema given the COMPANYDB
-- authorization identifier.  By design, several schema packages can be used
-- to define a single SQL schema.  This minimizes recompilation in that a
```

```
-- change to one schema package may not affect the other schema packages for
-- the same schema.  Programs referencing the schema do not require
-- recompilation unless they are dependent on the modified schema package.
-- Record type declarations in schema packages also declare database tables.
-- The record type name is used for the table name, component names are used
-- for the column names, and the component data types define the column data
-- types.  Note the specifications of the schema authorization identifier and
-- security classification definition, as discussed in comments on other
-- packages.

with SCHEMA_DEFINITION, COMPANYDB_AUTHORIZATION,
        COMPANYDB_TYPES, CLASSIFICATION_DEFINITION,
        COMPANYDB_TABLES_CLASSIFICATION;
 use SCHEMA_DEFINITION, COMPANYDB_AUTHORIZATION,
        COMPANYDB_TYPES, CLASSIFICATION_DEFINITION,
        COMPANYDB_TABLES_CLASSIFICATION;

package COMPANYDB_TABLES_SCHEMA is

  SCHEMA_AUTHORIZATION : IDENTIFIER := COMPANYDB;

  SECURITY : CLASSIFICATION := COMPANYDB_TABLES;

  type EMPLOYEE is
    record
      NAME : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
      AGE  : EMPLOYEE_AGE;
      SAL  : EMPLOYEE_SAL;
      DEPT : DEPT_NAME;
    end record;

  type DEPT is
    record
      NAME : DEPT_NAME_NOT_NULL_UNIQUE;
      LOC  : DEPT_LOC;
    end record;

end COMPANYDB_TABLES_SCHEMA;
-----------------------------------------------------------------------------
-- This is the body of the first COMPANYDB schema package.  Views, privileges,
-- and uniqueness constraints may be defined in package bodies.  This is done
-- by design, to minimize the number of recompilations required when adjusting
-- privileges or uniqueness constraints, or changing a view definition without
-- affecting the names or types of the columns returned.  Any of these changes
-- requires that only the affected package body be recompiled; it is not
-- necessary to recompile the package specification. Therefore, no other
-- recompilations are required.  COMPANYDB_TABLES is the package created by
-- the SQL function generator from the package specification of
-- COMPANYDB_TABLES_SCHEMA.  The required processing order for a schema
-- package is therefore:
-- (1) Compile the specification
-- (2) Run the SQL function generator on the specification
-- (3) Compile the body (one is required only if views and/or privileges are
--     defined, or if it is desired to define uniqueness constraints there)
-- (4) Run the schema translator after all specifications and bodies in a
--     schema have been compiled
-- Source files must be compiled before being given to the Ada/SQL automated tools.
```

```
-- Functions for database names are generated in COMPANYDB_TABLES.  The DEPT
-- and EMPLOYEE functions referenced here are among them.  A schema package must
-- have a name of the form X_SCHEMA, so that the package generated by the SQL
-- function generator may be named X.  This is the only package naming restriction
-- imposed by Ada/SQL.  Schema packages are related to authorization and classification
-- packages by calling the functions defined in those packages, not by package naming
-- convention.  It is, however, suggested that the example conventions of using
--  X_AUTHORIZATION and X_CLASSIFICATION as package names be continued.

with SCHEMA_DEFINITION, ADMINISTRATION_AUTHORIZATION,
         PERSONNEL_AUTHORIZATION, COMPANYDB_TABLES;
 use SCHEMA_DEFINITION, ADMINISTRATION_AUTHORIZATION,
         PERSONNEL_AUTHORIZATION, COMPANYDB_TABLES;

package body COMPANYDB_TABLES_SCHEMA is
begin

  GRANT ( SELEC, ON => DEPT,     TO => PUBLIC);
  GRANT ( ALLL,  ON => DEPT,     TO => ADMINSTRATION);
  GRANT ( ALLL,  ON => EMPLOYEE, TO => PERSONNEL);

end COMPANYDB_TABLES_SCHEMA;
-------------------------------------------------------------------------------
-- This is the classification package for the second COMPANYDB schema package.
-- This particular schema was segmented into a package for the base tables and
-- a package for the views.  This is not a requirement; as many base tables
-- and views as desired may be defined within the same schema package.

with CLASSIFICATION_DEFINITION;
 use CLASSIFICATION_DEFINITION;

package COMPANYDB_VIEWS_CLASSIFICATION is

  function COMPANYDB_VIEWS is new SECURITY_CLASSIFICATION
         (UNCLASSIFIED);

  type EMPVIEW is limited private;

private

  type EMPVIEW is
    record
      EMP  : CLASSIFICATION := UNCLASSIFIED;
      DEPT : CLASSIFICATION := UNCLASSIFIED;
    end record;

end COMPANYDB_VIEWS_CLASSIFICATION;
-------------------------------------------------------------------------------
-- This is the second schema package
-- Record type definitions are required for views as well as for base tables.
-- Views, however, are also defined in the bodies of schema packages.

with SCHEMA_DEFINITION, COMPANYDB_AUTHORIZATION,
        COMPANYDB_TYPES, CLASSIFICATION_DEFINITION,
        COMPANYDB_VIEWS_CLASSIFICATION;
  use SCHEMA_DEFINITION, COMPANYDB_AUTHORIZATION,
        COMPANYDB_TYPES, CLASSIFICATION_DEFINITION,
```

```
        COMPANYDB_VIEWS_CLASSIFICATION;

package COMPANYDB_VIEWS_SCHEMA is

  SCHEMA_AUTHORIZATION : IDENTIFIER := COMPANYDB;

  SECURITY : CLASSIFICATION := COMPANYDB_VIEWS;

  type EMPVIEW is
    record
      EMP  : EMPLOYEE_NAME;
      DEPT : DEPT_NAME;
    end record;

end
------------------------------------------------------------------------------
-- The body of the second COMPANYDB schema package with view and privilege
--  definitions COMPANYDB_TABLES is, as discussed before, generated from
-- COMPANYDB_TABLES_SCHEMA Likewise, COMPANYDB_VIEWS is generated
-- from the specification of this package Table and column names defined in
-- COMPANYDB_TABLES and referenced here are EMPLOYEE, NAME, and DEPT
-- Table and column names defined in COMPANYDB_VIEWS and referenced here are
-- EMPVIEW, EMP, and DEPT Note that DEPT is defined in both packages, and will
-- produce homographs.  DEPT is not a homograph when used to define the name of
-- an EMPVIEW column, since only COMPANYDB_VIEWS.DEPT can be used as such.
-- Using the appropriate definitions, Ada/SQL causes the Ada compiler to
-- require that column names used in a view definition in a package body must
-- have been declared as view columns in the corresponding package specific-
-- ation.  DEPT is, however, a homograph when used as an element in a SELEC
-- list, since both EMPVIEW.DEPT (defined in COMPANYDB_VIEWS) and
-- EMPLOYEE.DEPT (defined in COMPANYDB_TABLES) are valid column names.
-- Consequently, it must be qualified when used as such.  This is a hazard of
-- splitting a single schema into several portions homographs will arise if
-- the same name is defined in more than one schema package.  We have used the
-- SQL-style qualification -- EMPLOYEE.DEPT actually selects the component
-- named DEPT from the record returned by the function EMPLOYEE defined
-- in COMPANYDB_TABLES.  This is a different overloaded version of the EMPLOYEE
-- table than is called on the next line.  An Ada-style qualification could
-- have also been used COMPANYDB_TABLES.DEPT would select the DEPT column
-- function from the COMPANYDB_TABLES package.  Duplicate column names within
-- the same schema package do not cause homographs only one column function
-- (actually a set of overloaded functions) is defined.  Due to the
-- implementation of strong typing, however, the use of an unqualified column
-- name in the same expression as a literal, universal, or overloaded value
-- may cause an unresolvable ambiguity if there exist columns of different types
-- having that same name.  The ambiguity can be resolved by qualifying the
-- column name with the appropriate table name. In this example, NAME is such a
-- potentially ambiguous column name. Its use in the SELEC list is,
-- however, not ambiguous.  (We are here talking about ambiguity to the
-- Ada compiler.  Qualification with the table name may be required
-- only to remove the Ada ambiguity;  the unqualified column name may
-- be unambiguous to SQL.)

with SCHEMA_DEFINITION, COMPANYDB_TABLES, COMPANYDB_VIEWS;
 use SCHEMA_DEFINITION, COMPANYDB_TABLES, COMPANYDB_VIEWS;

package body COMPANYDB_VIEWS_SCHEMA is
```

```
begin

  CREATE_VIEW ( EMPVIEW (  EMP  & DEPT ),
          AS => SELEC   (  NAME & EMPLOYEE.DEPT,
                FROM   =>  EMPLOYEE ) );

  GRANT ( SELEC, ON => EMPVIEW, TO => PUBLIC);

end COMPANYDB_VIEWS_SCHEMA;
```

Note: This example demonstrates a possible Ada/SQL definition of the illustrative database used within the paper "Proposed Language Access to Draft Proposed American National Standard Database Language SQL", ANSC X3H2 (Database), March 1985, which is defined in SQL as:

```
SCHEMA
AUTHORIZATION COMPANYDB
TABLE EMPLOYEE
      (NAME  CHARACTER(15)  NOT NULL UNIQUE,
       AGE      NUMERIC(3,0),
       SAL       DECIMAL(8,2),
       DEPT    CHARACTER(10))
TABLE DEPT
      (NAME  CHARACTER(10)  NOT NULL UNIQUE,
       LOC      CHARACTER (2))
VIEW EMPVIEW (EMP, DEPT)
   AS SELECT  NAME, DEPT   FROM  EMPLOYEE
GRANT SELECT  ON  DEPT  TO  PUBLIC
GRANT SELECT  ON  EMPVIEW  TO  PUBLIC
GRANT ALL  ON  DEPT  TO  ADMINISTRATION
GRANT ALL  ON  EMPLOYEE  TO  PERSONNEL
```

FORMAT:

<schema> ::= <compilation unit> ...

Several Ada compilation units, all packages, combine together to form a schema. Building a single schema out of several packages adheres to the Ada modular program philosophy, and allows parts of schemas to be modified without necessarily requiring recompilation of all programs using the schema.

```
<compilation unit> ::=
     <context clause> <library unit> | <context clause> <secondary unit>

<context clause> ::= as in Ada, except that only packages may be named
```

The packages comprising a schema can "with" and "use" other packages as required for visibility.

```
<library unit> ::=
       <authorization package declaration>
     | <classification package declaration>
     | <schema package declaration>

<secondary unit> ::= <library unit body>

<library unit body> ::= <schema package body>
```

Four types of compilation units are used within schemas:

(1) Authorization packages declare authorization identifiers.

(2) Classification packages declare the classification of all columns defined within the corresponding schema package declaration.

(3) Schema packages declare database tables and columns, and

(4) The bodies of schema packages declare views, grant privileges, and may set uniqueness constraints.

<authorization package declaration> ::= see section 4.1a

<classification package declaration> ::= see section 4.6a

<schema package declaration> ::= see section 4.1b

<schema package body> ::= see section 4.5

4.1a <schema> - <authorization package declaration>

FUNCTION:

Each authorization package declares a different authorization identifier. This authorization identifier may be used as a schema authorization identifier and/or as a target identifier for granting privileges.

EXAMPLE:

```
with SCHEMA_DEFINITION;
  use SCHEMA_DEFINITION;

package ADMINISTRATION_AUTHORIZATION is

  function ADMINISTRATION is new AUTHORIZATION_IDENTIFIER;

end ADMINISTRATION_AUTHORIZATION;
```

FORMAT:

```
<authorization package declaration> ::=
     <authorization package specification> ;

<authorization package specification> ::=
  package <identifier> is
    function <authorization identifier> is new AUTHORIZATION_IDENTIFIER;
  end [ <package simple name> ]

<identifier> ::= any valid name for a library package

<authorization  identifier> ::= the identifier that will be used for
     granting privileges and/or as a schema authorization identifier

<package simple name> ::= must match the package <identifier> if used
```

To define the generic function AUTHORIZATION_IDENTIFIER, the context clause of an authorization package must read:

```
with SCHEMA_DEFINITION; use SCHEMA_DEFINITION;
```

The sole function of an authorization package is to define an authorization identifier. Authorization packages do not have bodies.

4.1b <schema> - <schema package declaration>

FUNCTION:

Schema package declarations define the table and column names within a schema.

EXAMPLE:

```
with SCHEMA_DEFINITION, COMPANYDB_AUTHORIZATION,
        COMPANYDB_TYPES, CLASSIFICATION_DEFINITION,
        COMPANYDB_VIEWS_CLASSIFICATION;
  use SCHEMA_DEFINITION, COMPANYDB_AUTHORIZATION,
        COMPANYDB_TYPES, CLASSIFICATION_DEFINITION,
        COMPANYDB_VIEWS_CLASSIFICATION;

package COMPANYDB_VIEWS_SCHEMA is

  SCHEMA_AUTHORIZATION : IDENTIFIER := COMPANYDB;

  SECURITY : CLASSIFICATION := COMPANYDB_VIEWS;

  type EMPVIEW is
    record
      EMP    : EMPLOYEE_NAME;
      DEPT   : DEPT_NAME;
    end record;

end COMPANYDB_VIEWS_SCHEMA;
```

FORMAT:

<schema package declaration> ::= <schema package specification> ;

```
<schema package specification> ::=
     package <identifier> is
       <schema authorization clause>
     [ <schema classification clause> ]
     [ <schema declaration element> ... ]
     end [ <package simple name> ]
```

A <schema classification clause> is only required by the MIL-STD, not by the ANSI standard. If all tables declared within the schema package are views, then a <schema classification clause> is optional. In the absence of a classification specification for a view, the classification of each column defaults to the most restrictive classification on the data used to materialize that column.

```
<identifier> ::= a valid name for a library package, that must be of
     the form X_SCHEMA to permit the package generated from this one to
     be named X  by the SQL function generator

<package simple name> ::= must match the package <identifier> if used

<schema authorization clause> ::=
     SCHEMA_AUTHORIZATION : [ SCHEMA_DEFINITION . ] IDENTIFIER
          := <schema authorization identifier> ;

<schema authorization identifier> ::= <authorization identifier>

<authorization identifier> ::= as defined within an authorization
     package (see section 4.1a)
```

The SQL <schema authorization identifier> for the schema is taken to be the <authorization identifier> used here. The

appropriate authorization package must, of course, be named within the context clause of the schema package in order to make the instantiated function for the <authorization identifier> visible. The full name of the <authorization identifier> may be used if visibility is desired by selection: only the simple name is used as the <authorization identifier>. SCHEMA_DEFINITION must also be named within the context clause to make the type IDENTIFIER visible. As indicated, IDENTIFIER may be visible either directly or by selection. The same <authorization identifier> may be referenced from several schema packages; the tables declared in those packages are all placed in the same SQL schema.

```
<schema classification clause> ::=
     SECURITY : [ CLASSIFICATION_DEFINITION . ] CLASSIFICATION
          := <classification identifier> ;

<classification  identifier>  ::= as defined within a classification
     package (see section 4.6a)
```

The appropriate classification package must, of course, be named within the context clause of the schema package in order to make the instantiated function for the <classification identifier> visible. The full name of the <classification identifier> may be used if visibility is desired by selection. CLASSIFICATION_DEFINITION must also be named within the context clause to make the type CLASSIFICATION visible. As indicated, CLASSIFICATION may be visible either directly or by selection.

```
<schema declaration element> ::=
     <basic declarative item> | <table definition>

<basic  declarative item> ::= an Ada basic declarative item,  subject
     to the interpretations and restrictions discussed in section 4.7

<table definition> ::= see section 4.2
```

Declarations within schema packages are not limited to the definition of database tables: types, etc. may also be defined as required.

4.2 <table definition>)

FUNCTION:

Define the name of a database table, as well as the column names and data types. Unlike SQL, table definitions must be given for both base tables and views. A view is distinguished from a base table by having a view definition in the body of the schema package in which the corresponding table definition appears.

EXAMPLE:

```
type EMPVIEW is
  record
    EMP  : EMPLOYEE_NAME;
    DEPT : DEPT_NAME;
  end record;
```

FORMAT:

```
<table definition> ::=
     type <table name> [ <discriminant part> ] is
       record
         <component list>
       end record;

<table name> ::= a valid Ada record type name,  which is also taken to
     be the name of the database table being declared
```

Within a schema package, the declaration of a record type that is not referenced from another record type also declares a database table. The name of the table is taken to be the name of the record type, and the columns of the table are given by the components of the record type. "Columns" may be of composite types, so lowest level subcomponents become actual database columns. Ada/SQL does, however, permit access to data as logically structured and grouped by composite types. The runtime system will translate between Ada composite structures and the simpler underlying database representation. Subcolumns may also be accessed by using indexing or component selection for array and record columns, respectively.

```
<discriminant part> ::=
     ( <discriminant table element>
          [ { ; <discriminant table element> } ... ] )

<discriminant table element> ::= <table element>

<component list> ::=
       <table element> ; [ { <table element> ; } ... ]
     | [ { <table element> ; } ... ] <variant part>
     | null ;
```

Both discriminant parts and component lists declare record components, and hence table columns. In Ada, a discriminant specification (<discriminant table element> here) looks essentially like a component declaration (<table element> here), with the following restrictions: (1) The type/subtype of a discriminant may only be given by a type mark, not by a general subtype indication, and (2) discriminants must be discrete. These restrictions must be remembered when reading the following discussion of <table element>s -- they will be enforced by the Ada compiler and by the Ada/SQL automated tools.

No corresponding database table is defined for the declaration of a null record.

```
<variant part> ::=
     case <discriminant simple name> is
          <variant>
        [ <variant> ... ]
     end case ;

<discriminant simple name> ::= as in Ada

<variant> ::=
```

```
        when <choice> [ {    <choice> } ... ] =>
          <component list>

    <choice> ::= as in Ada
```

Accessing a column within a variant part restricts the rows that are considered to be within the table. In particular, only those rows that would contain the column, based on the Ada meaning of the discriminant value, are accessed.

```
    <table element> ::=
           <column definition>
         | <single column unique constraint definition>

    <column definition> ::= see section 4.3

    <single column unique constraint definition> ::= see section 4.4
```

4.3 <column definition>

FUNCTION:

Define the names, data types, null value possibilities, and uniqueness requirements for database columns.

EXAMPLE:

```
NAME : EMPLOYEE_NAME_NOT_NULL_UNIQUE;
AGE  : EMPLOYEE_AGE;
```

FORMAT:

```
<column definition> ::=
     <column name list> : <data type> [ _NOT_NULL [ _UNIQUE ] ]
          [ <constraint> ] [ := <expression> ]
```

```
<column name list> := n Ada identifier list -- the Ada syntax for
     discriminant specifications and component declarations permits
     several columns to be defined within the same <column definition>.
     The component names and data types are used for the column names
     and data types.
```

```
<data type> ::= an Ada type mark. The type mark may include the
     suffixes _NOT_NULL or _NOT_NULL_UNIQUE, which causes the
     appropriate constraint to be defined for the database.
```

```
<constraint> ::= an Ada constraint, permitted only if a component other
     than a discriminant is being defined. The constraint must be
     static, unless it depends on a discriminant, in which case
     discriminant names may be the only non-static items in the
     constraint.
```

```
<expression> ::= an Ada expression that provides default values for the
     record components and also for the database columns. When insert-
     ing rows with unspecified values for columns that have defaults,
     the default values are stored in the columns. Null values are
     stored in columns with neither specified values nor defaults, with
     an error occurring if any of those columns do not permit null val-
     ues. The default values used for a table are the same as would be
     used by Ada when creating an object of the record type correspond-
     ing to the table, at the point in the program of the insert opera-
     tion. Default expressions must be static in Ada/SQL, since they
     are processed at compile time, except that the name of a discrim-
     inant may be used.
```

4.4 <unique constraint definition>

FUNCTION:

Specify that a column or a group of columns is to contain only unique data.

EXAMPLE:

```
NAME : DEPT_NAME_NOT_NULL_UNIQUE; -- 1

CONSTRAINTS ( EMPLOYEE , UNIQUE ( SAL & DEPT ) ); -- 2
```

FORMAT:

```
<unique constraint definition> ::=
        <single column unique constraint definition>
      | <multiple column unique constraint definition>
```

A <single column unique constraint definition> is used to apply a uniqueness constraint to a single column, directly as the column is defined in its enclosing <table definition> (see section 4.2). Applying a <single column unique constraint definition> to a composite column defines a uniqueness constraint over several underlying database columns. A <multiple column unique constraint definition> is used to apply a uniqueness constraint on several Ada/SQL columns. (It may also be used with single columns.)

<single column unique constraint definition> ::= <column definition>

<column definition> ::= see section 4.3

No special Ada/SQL <single column uniqueness constraint definition> syntax is required -- Ada/SQL column definitions provide all the capabilities necessary to define single column uniqueness constraints. Suffixing a column's data type with _NOT_NULL_UNIQUE defines a uniqueness constraint, as shown in example 1. If the column is a composite column, the uniqueness constraint is actually over a group of underlying database columns, which is part of the capability provided by SQL <unique constraint definition>s. The remaining capability provided by SQL <unique constraint definition>s, that of including the same database column in several <unique constraint definition>s, is provided by <multiple column unique constraint definition>s in Ada/SQL.

```
<multiple column unique constraint definition> ::=
     CONSTRAINTS ( <table name> , UNIQUE ( <unique column list> ) ) ;
```

A <multiple column unique constraint definition>, as shown in example 2, is placed in the <schema package body> (see section 4.5) corresponding to the <schema package declaration> in which the referenced <table name> is defined. Its effect is to cause CONSTRAINTS UNIQUE(<unique column list>) to be added to the SQL <table definition> for the <table name>. Consequently, it may be applied only to a base table.

```
<table name> ::= the name of a table defined in the corresponding schema
     package declaration.  The functions for the table name are defined
     in the package produced by the SQL function generator from the
     schema package declaration, as are the table-specific CONSTRAINTS
     procedure and  UNIQUE function.

<unique column list> ::=
     <column name> [ { & <column name> } ... ]
```

SQL uses commas to separate the elements of a <unique column list>; Ada/SQL uses ampersands. The ampersand functions are defined specifically for each table by the SQL function generator.

```
<column name> ::= a name of a column defined for the table. The
     functions for the column names are defined by the SQL function
     generator.
```

4.5 <schema package body>

FUNCTION:

Views, privileges, and multiple column uniqueness constraints are defined within the bodies of schema packages.

EXAMPLE:

```
package body COMPANYDB_VIEWS_SCHEMA is
begin

  CREATE_VIEW ( EMPVIEW ( EMP  & DEPT ),
          AS => SELEC   ( NAME & EMPLOYEE.DEPT,
                FROM    => EMPLOYEE ) );

  GRANT ( SELEC, ON => EMPVIEW, TO => PUBLIC);

end COMPANYDB_VIEWS_SCHEMA;
```

FORMAT:

```
<schema package body> ::=
     package body <package simple name> is
       [ <declarative part> ]
     begin
       <schema body element> ...
     end [ <package simple name> ] ;

<package simple name> ::= the package identifier of the corresponding
    schema package declaration.  Schema packages that do not define
    views, grant  privileges, or require multiple column uniqueness
    constraints are not  required to have bodies.
<declarative part> ::= as in Ada, except that the only declarations per-
    mitted are static constant object declarations, number declarations,
    renaming declarations, and use clauses. In short, only declarations
    that will add to the convenience of defining views, privileges, and
    multiple column uniqueness constraints are permitted.

<schema body element> ::=
     <view definition>  |  <privilege definition>
    | <multiple column unique constraint definition>

<view definition> ::= see section 4.5a

<privilege definition> ::= see section 4.6

<multiple column unique constraint definition> ::= see section 4.4
```

For every view definition or multiple column unique constraint definition in a schema package body, there must be a corresponding table definition in the corresponding schema package declaration.

4.5a <view definition>

FUNCTION:

Define a viewed table.

EXAMPLE:

```
CREATE_VIEW ( EMPVIEW ( EMP  & DEPT ),
        AS => SELEC   ( NAME & EMPLOYEE.DEPT,
              FROM   => EMPLOYEE ) );
```

FORMAT:

```
<view definition> ::=
     CREATE_VIEW ( <table name> [ ( <view column list> ) ] ,
            AS => <query specification> [ ,
            WITH_CHECK_OPTION => ENABLED ] ) ;
```

The two SQL keywords CREATE and VIEW are linked into a single Ada identifier with an underscore. The CREATE_VIEW procedure is defined within SCHEMA_DEFINITION.

```
<table name> ::= the name of a table defined in the corresponding schema
   package declaration.  The functions for the table name are defined
   in the package produced by the SQL function generator from the schema
   package declaration.  If a <view column list> is included, and the
   <table name> includes an <authorization identifier>, then the <table
   name> must be expressed as <authorization identifier>-<table
   identifier>, instead of as <authorization identifier>.<table
   identifier>, as used most elsewhere within Ada/SQL.

<view column list> ::=
   <column name> [ { & <column name> } ... ]

<column name> ::= a name of a column defined for the table. If a view
   column list is given, the column names must agree precisely, in
   order, with those defined for the table in the schema package dec-
   laration.  The functions for the column names are also defined in
   the package produced by the SQL function generator from the schema
   package declaration.

<query specification> ::= see section 3.25.  The number of columns def-
    ined for the view must be the same as the degree of the table speci-
    fied by the <query specification>. Furthermore, the data type of
    each column of the <query specification> must be capable of being
    converted, via an Ada type conversion, to the data type of the
    corresponding column defined for the view.
```

4.6 <privilege definition>

FUNCTION:

Define privileges.

EXAMPLE:

```
GRANT ( SELEC, ON => EMPVIEW, TO => PUBLIC);
GRANT ( ALLL,  ON => DEPT,    TO => ADMINISTRATION);
```

FORMAT:

```
<privilege definition> ::=
     GRANT ( <privileges> ,
       ON => <table name> ,
       TO => <grantee> [ { & <grantee> } ... ] [ ,
       WITH_GRANT_OPTION => ENABLED ] ) ;
```

GRANT procedures and ancillary functions for each table defined in a schema package declaration are defined in the package produced from that schema package declaration by the SQL function generator. Ancillary functions that are not table-specific are defined in SCHEMA_DEFINITION. Each clause in the GRANT statement is a parameter to the GRANT procedure. Parameter names retain the SQL keywords. WITH_GRANT_OPTION differs slightly from SQL to be more Ada-like, and is an optional last parameter. The grantee list is linked together with ampersands; in SQL there are no operators between the grantees.

```
<privileges> ::=
       ALLL  |  ALL_PRIVILEGES
     | <action> [ { & <action> } ... ]
```

ALL is an Ada reserved word and so cannot be used, and ALL PRIVILEGES is connected with an underscore. In SQL the action list is separated by commas, Ada/SQL uses ampersands as elsewhere. ALLL and ALL_PRIVILEGES are functions defined in SCHEMA_DEFINITION.

```
<action> ::=
       SELEC  |  INSERT  |  DELETE
     | UPDATE [ ( <grant column list> ) ]
```

SQL uses SELECT; Ada/SQL uses SELEC because SELECT is an Ada reserved word. The SELEC, INSERT, DELETE, and UPDATE functions, as they apply to privilege definitions, are defined in the appropriate generated package.

```
<grant column list> ::=
     <column name> [ { & <column name> } ... ]
```

The column names are linked together with ampersands rather than commas, as is customary within Ada/SQL.

```
<column name> ::= the name of a column in the table for which privileges
     are being granted.  Functions defining column names are produced
     from schema package declarations by the SQL function generator.

<grantee> ::=
     PUBLIC  |  <authorization identifier>

<authorization identifier> ::= as defined within an authorization
     package (see section 4.1a)
```

A function defining PUBLIC is declared in SCHEMA_DEFINITION. Authorization identifiers are defined in the applicable authorization packages, which must be named within the context clause of the schema package body in order to be used.

4.6a <classification package declaration>

FUNCTION:

Classification package declarations are used to define the classifications of all columns of the database. Each schema package declaring base tables must have a corresponding classification package. The classification package is "with"ed into the schema package, and a <schema classification clause> (see section 4.1b) is used to indicate the relation of the classification package to the schema package. Classification packages are required only by the MIL-STD, not by the ANSI standard.

EXAMPLE:

```
package COMPANYDB_VIEWS_CLASSIFICATION is

  function COMPANYDB_VIEWS is new SECURITY_CLASSIFICATION(UNCLASSIFIED);

  type EMPVIEW is limited private;

private

  type EMPVIEW is
    record
      EMP  : CLASSIFICATION := UNCLASSIFIED;
      DEPT : CLASSIFICATION := UNCLASSIFIED;
    end record;

end COMPANYDB_VIEWS_CLASSIFICATION;
```

FORMAT:

```
<classification package declaration> ::=
     <classification package specification> ;

<classification package specification> ::=
     package <identifier> is
       function <classification identifier> is new
         [ CLASSIFICATION_DEFINITION . ] SECURITY_CLASSIFICATION
             ( <classification> ) ;
       <private type declaration> ...
     private
       <table definition> ...
     end [ <package simple name> ]
```

The <classification identifier> is defined by instantiating the generic SECURITY_CLASSIFICATION function which is defined within CLASSIFICATION_DEFINITION. As indicated, SECURITY_CLASSIFICATION can be made visible either directly or by selection.

```
<identifier> ::= any valid name for a library package

<package simple name> ::= must match the package <identifier> if used

<classification identifier> ::= the identifier that will be used in the
     <schema classification clause> of a schema package to indicate that
     this classification package declaration is applicable to it (see
     section 4.1b)

<classification> ::= a value of type CLASSIFICATION, indicating the most
     restrictive classification applied within this classification pack-
     age declaration. Type CLASSIFICATION is defined within CLASSIFICA-
     TION_DEFINITION, and may be tailored for the specific environment
     to include basic classification, releasability, special handling,
```

and sensitive source information.

<private type declaration> ::= as in Ada, with a limited private type declaration for each table definition in the schema package that will use this classification package declaration. The names and discriminants of the limited private types are the same as those of the corresponding record types. (This duplication of record type names is the reason that separate classification packages are required.) All discriminants are of type CLASSIFICATION, however. An additional discriminant, called TABLE, may be defined if it is desired to specify a classification for the entire table. In the absence of the special TABLE discriminant, a table is given the most restrictive classification defined for all of its columns. Each discriminant must be given a default value, which indicates the classification of the corresponding column.

<table definition> ::= see section 4.2. Corresponding full type declarations must be provided for all limited private types declared. The full type declarations must have components named the same, in the same order, as the corresponding table definitions to which they apply. All components are of type CLASSIFICATION, however. Default values must be provided for all components, to indicate the classification of the corresponding database columns. It should be noted that composite columns have but one classification; later versions of this standard may address separate classifications for subcolumns.

4.7 Special Considerations

This section describes special considerations that apply to data defined within Ada/SQL schemas. Subsections are organized according to the section numbering and notational conventions of the Ada Programming Language Military Standard (ANSI/MIL-STD-1815A), hereafter referred to as the Language Reference Manual (LRM).

4.7 Special Considerations - LRM section 2.3 - identifier

FUNCTION:

Identifiers perform their usual Ada functions within schemas, but are also used to name SQL authorization identifiers, tables, and columns.

```
EXAMPLE:
CITY                 -- columns of type/subtype CITY may contain null
                     -- and/or duplicate values
CITY_NOT_NULL        -- columns of this type/subtype may not contain
                     -- null values, but may contain duplicate values
CITY_NOT_NULL_UNIQUE -- columns of this type/subtype may not contain
                     -- nulls and also may not contain duplicate values
```

FORMAT:

```
identifier ::= as in Ada
```

Any legal Ada identifiers may be used within schemas. Identifiers used as SQL authorization identifiers, table names, or column names may not be passed to the database management system, however, because (1) SQL identifiers may include only upper case characters (case is not significant in Ada anyway), (2) SQL identifiers may not be longer than 18 characters, (3) SQL identifiers may not be identical to SQL key words, and (4) certain Ada/SQL data definitions may cause underlying database tables to have duplicate table and/or column names unless the names are qualified by selection in the Ada sense. Ada/SQL maps Ada identifiers (and full names, where necessary to avoid duplicates) to appropriate underlying SQL identifiers, with an algorithm that attempts to maintain as much semantic content within names as possible.

Identifiers used as type and subtype names may include the suffixes _NOT_NULL or _NOT_NULL_UNIQUE. Database columns defined by record subcomponents of types/subtypes named with these suffixes will be given the corresponding SQL constraints.

A restriction on the use of the _NOT_NULL and _NOT_NULL_UNIQUE suffixes ensures that only closely related types/subtypes have similar simple names. Two identifiers are considered similar if they differ only in the use or nonuse of the _NOT_NULL and _NOT_NULL_UNIQUE suffixes. If A and B are two directly visible types/subtypes with similar simple names, then one of the following definitions must hold, where C is another type, not necessarily directly visible, with simple name similar to those of A and B:

```
(1) subtype A is B;     (3) subtype C is A;     (4) subtype C is B;
(2) subtype B is A;         subtype B is C;         subtype A is C;
```

4.7 Special Considerations - LRM section 3.1 - declaration

FUNCTION:

Declare the tables and columns of a database, as well as the types of data to be stored within the database.

EXAMPLE:

```
SECURITY_MARKING_LENGTH : constant := 12; -- a number declaration

type SECURITY_CLASSIFICATION is (U,C,S,T); -- type declarations
type SECURITY_MARKING is new STRING(1..SECURITY_MARKING_LENGTH);

MINIMUM_CLASSIFICATION : constant SECURITY_CLASSIFICATION := C;
  -- an object declaration

MINCLASS : SECURITY_CLASSIFICATION renames a MINIMUM_CLASSIFICATION;
  -- a renaming declaration

subtype CLASSIFICATION_LEVEL is SECURITY_CLASSIFICATION
  range MINIMUM_CLASSIFICATION..SECURITY_CLASSIFICATION'LAST;
    -- a subtype declaration

type MARKING_TABLE is                  -- a record type declaration that
  record                                -- also serves to declare a data-
    CLASS : SECURITY_CLASSIFICATION;   -- base table and its columns
    MARK  : SECURITY_MARKING;           -- (the table in this example is
  end record;                          -- rather trivial and unecess-
                                        -- sary)
```

FORMAT:

```
basic_declaration ::= as in Ada, except that the only declarations perm-
    itted within schema packages are object declarations (see LRM sect-
    ion 3.2), number declarations (see LRM section 3.2), type declara-
    tions (see LRM section 3.3.1), subtype declarations (see LRM section
    3.3.2) and renaming declarations (see LRM section 8.5).
```

The only declarations that are permitted within a schema package are those that apply directly to the data definition function. Further restrictions are placed on the various declarations, as discussed in the appropriate sections following. Packages named in the context clause of schema packages may contain arbitrary declarations, if they are not also schema packages.

4.7 Special Considerations - LRM section 3.2 - constant object and named number

FUNCTION:

Define named constants (of arbitrary type) and numbers (of numeric type)

EXAMPLE:

```
LIMIT      : constant INTEGER := 10_000;    -- taken from LRM section
LOW_LIMIT  : constant INTEGER := LIMIT/10;  -- 3.2.1 in that example,
                                     -- TOLERANCE is not static and hence
PI         : constant := 3.14159_26536;  -- may not be used in a schema
TWO_PI     : constant := 2.0*PI;         -- number declaration examples
MAX        : constant := 500;            -- are taken from LRM section
POWER_16   : constant := 2**16;          -- 3.2.2
ONE, UN, EINS     : constant := 1;
```

FORMAT:

```
object_declaration ::= as in Ada, except that only constants may be dec-
     lared in a schema package. Likewise, objects referenced from schema
     packages, except for discriminants, must be constants.

number_declaration ::= as in Ada
```

The purpose of a schema package is to declare database objects, not program objects. Hence, the declaration of arbitrary objects is not permitted within schema packages. Constants and named numbers are, however, permitted as a convenience, to allow meaningful names to be given to important values. All entities used within expressions in schema packages, except for discriminants, must be static, because all Ada/SQL automated tool processing is performed at compile time.

4.7 Special Considerations - LRM section 3.3.1 - type declaration

FUNCTION:

Declare types of program and database values. Also, indicate names and columns of database tables.

EXAMPLE:

```
type DIGIT_CHARACTER is ('0','1','2','3','4','5','6','7','8','9');
  -- an enumeration type

type DIGIT_VALUE is range 0..9; -- an integer type

type MONEY is delta 0.01 range 0.00..1_000_000.00; -- a real type

type PHONE_NUMBER is array (1..10) of DIGIT_CHARACTER; -- an array type

type PHONE_BILL is -- a record type (also a database table in this
  record           -- example)
    NUMBER : PHONE_NUMBER;
    DUE    : MONEY;
  end record;

type OVERDUE_PHONE_BILL is new PHONE_BILL;
  -- a derived type (also a database table in this example)
```

FORMAT:

```
type_declaration ::= as in Ada, except that incomplete type declara-
   tions, private type declarations, access type definitions, and task
   type declarations are not permitted within schema packages.  The
   following types may therefore be declared within schema packages:
   enumeration (see LRM section 3.5.1), integer (see LRM section 3.5.4),
   real (see LRM section 3.5.6), array (see LRM section 3.6), record
   (see LRM section 3.7), and derived (see LRM section 3.4).
```

Access and task types may not be declared within schema packages. Subcomponents of composite types declared or used within schema packages may also not be of an access or task type. As a result, incomplete type declarations are not necessary, and may not be used, within schema packages.

Private types may not be declared within schema packages, since their declaration would normally also require the declaration of subprograms defining operations on the types. And subprogram declarations are not permitted within schema packages, in order to restrict package text to that required specifically for the data definition function. However, subcomponents of record and array types may be of a private type (defined in other library units used by the schema), providing that the corresponding full type would be permitted to be used for the subcomponents.

The Ada/SQL operations available on a database column of a private type are limited to the following:

(1) Equality (EQ) and inequality (NE) comparisons, including within <quantified predicate>s,

(2) Use as the argument of a COUNT_DISTINCT set function,

(3) IS_IN and NOT_IN comparisons,

(4) IS_NULL and IS_NOT_NULL tests (a composite column is null if and only if all subcolumns are null),

(5) Use in a GROUP_BY clause,

(6) Use in the SELEC clause of a <sub-query>, <query specification>, or <select statement>,

(7) Use in the <insert column list> of an INSERT_INTO statement.

(8) Use within a <set clause> of either type of UPDATE statement, as either the <object column> or the <value expression>.

(9) Use within a <grant column list> of a GRANT statement, and

(10) Selection of components that are discriminants.

Further restriction of SQL operations to develop an analogue to limited private types is not fruitful; hence, limited private types may not be used for database columns.

There are some other considerations that apply to the various classes of type declaration within schemas; these are discussed within the referenced section for each class of type. Any type that may be declared within a schema package can be used as the type of a database column, and no other types may be so used.

4.7 Special Considerations - LRM section 3.3.2 - subtype declaration

FUNCTION:

Declare Ada subtypes. Subtypes may be used in their Ada sense, and to indicate _NOT_NULL and _NOT_NULL_UNIQUE constraints. Although not a part of this standard, the way in which subtypes relate to each other can also be used to guide test data generation.

EXAMPLE:

```
type EMPLOYEE_NUMBER is range 1000..9999;

subtype EMPLOYEE_NUMBER_NOT_NULL_UNIQUE is EMPLOYEE_NUMBER;
```

FORMAT:

```
subtype_declaration ::= as in Ada, except that it must be static, and
    may only define a subtype that would be permitted for a database
    column
```

Because the Ada/SQL automated tools will process schemas at compile time, all subtypes defined must be static. Furthermore, only subtypes that would be legal for database columns may be defined within a schema.

A subtype of a _NOT_NULL or _NOT_NULL_UNIQUE type or subtype does not inherit these properties: they are determined solely based on the suffix of the type or subtype name.

4.7 Special Considerations - LRM section 3.4 - derived type definition

FUNCTION:

Define a new type with characteristics derived from another type.

EXAMPLE:

```
type PROMOTION_LIST_MEMBER is new EMPLOYEE_NUMBER;
  -- derived from the example on LRM section 3.3.2, above
```

FORMAT:

```
derived_type_definition ::= as in Ada, except that it must be static,
     and may only define a subtype that would be permitted for a
     database column
```

The type mark in the subtype indication of a derived type definition may include the suffix _NOT_NULL or _NOT_NULL_UNIQUE. However, these properties do not carry over to the derived type; they are only conveyed by the suffix on a type or subtype name.

If the derived type is a record type, then the derived type declaration also declares a database table according to the same rules as for other record types: The derived type declaration also declares a database table if and only if the name of the derived type (or any subtype with a name differing only in the _NOT_NULL or _NOT_NULL_UNIQUE suffix) is not used as the type of a subcomponent within another record declaration within the same schema package. A derived database table is assumed to be a base table unless a view definition is given for it.

Type representation clauses, as they affect database representation, apply to derived types as with Ada: A type representation clause for the parent type applies to the derived type if and only if it appears before the declaration of the derived type.

4.7 Special Considerations - LRM section 3.5.1 - enumeration type definition

FUNCTION:

Define an enumeration type and its values.

EXAMPLE:

```
type DAY    is (MON, TUE, WED, THU, FRI, SAT, SUN); -- examples taken
type SUIT   is (CLUBS, DIAMONDS, HEARTS, SPADES);  -- from LRM sections
type GENDER is (M, F);                             -- 3.5.1 and 3.5.2
type LEVEL  is (LOW, MEDIUM, URGENT);
type COLOR  is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
type LIGHT  is (RED, AMBER, GREEN);

type HEXA   is ('A', 'B', 'C', 'D', 'E', 'F');
type MIXED  is ('A', 'B', '*', B, NONE, '?', '*');

type ROMAN_DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```

FORMAT

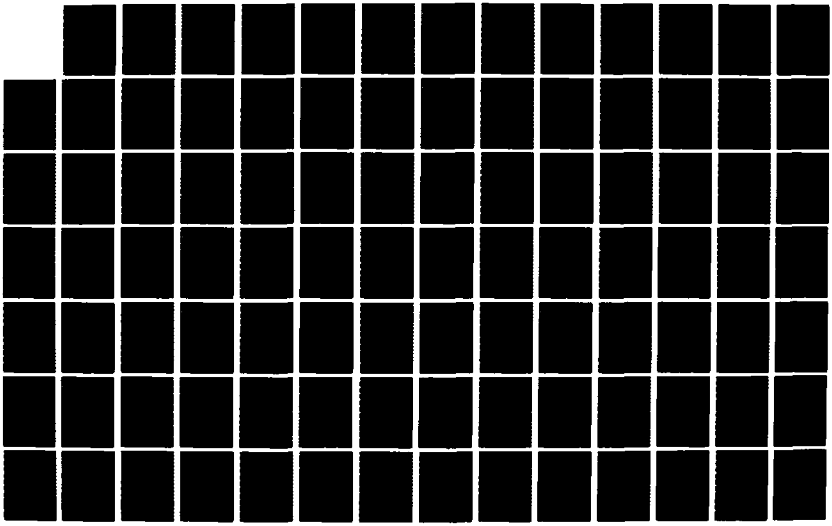
```
enumeration_type_definition ::= as in Ada
```

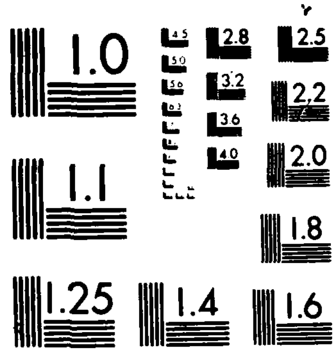
The Ada/SQL automated tools will recognize the predefined types CHARACTER and BOOLEAN. The SQL operations available on enumeration types are the same as are available on strings, except that LIKE is not available for enumeration types.

SQL does not directly support enumeration types, so it is necessary to map Ada enumeration types into other SQL data types. The mapping process should achieve three objectives: (1) preserve the ordering of the enumeration values, (2) use explicit representations if requested, and (3) enable enumeration database values to be referenced/retrieved by their identifiers or character literals, even from non-Ada ways of accessing the database.

For any database type, subtype, or table T, T_NOT_NULL, or T_NOT_NULL_UNIQUE, the SQL function generator defines a named number T_SIZE, of type universal_integer, to be equal to the minimum number of bits that is needed by the DBMS implementation to hold any possible object of this type or subtype. This is defined as in Ada, except considering DBMS storage units and representation instead of those of the host computer. Values for private types are as would be given for the underlying full type.

AD-A182 885 PRELIMINARY VERSION· ADA (TRADE NAME)/SQL· A STANDARD 2/3
PORTABLE ADA-DBMS I (U) INSTITUTE FOR DEFENSE ANALYSES
ALEXANDRIA VA B R BRYKCZYNSKI ET AL APR 87 IDA-P-1944
UNCLASSIFIED IDA/HQ-86-31239 MDA903-84-C-0031 F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

4.7 Special Considerations - LRM section 3.5.4 - integer type definition

FUNCTION:

Define the range of values for an integer type.

EXAMPLE:

```
type PAGE_NUM  is range 1..2_000;   -- taken from LRM section 3.5.4
type LINE_SIZE is range 1..MAX_LINE_SIZE;

subtype SMALL_INT   is INTEGER   range -10..10;
subtype COLUMN_PTR  is LINE_SIZE range 1..10;
subtype BUFFER_SIZE is INTEGER   range 0..MAX;
```

FORMAT:

```
integer_type_definition ::= as in Ada, except that it must be static
```

The Ada/SQL automated tools will recognize the predefined type INTEGER. Although implementation-specific type names, such as LONG_INTEGER and SHORT_INTEGER, will not be recognized, range constraints can be used to define types with the corresponding ranges.

The range of integers supported by a database management system may not be the same as that supported by the Ada system used to access that DBMS. The package DATABASE provides information about the DBMS being accessed by Ada/SQL. In particular, the smallest (most negative) integer value supported by the DBMS through Ada/SQL is the named number DATABASE.MIN_INT and the largest (most positive) value is DATABASE.MAX_INT. The type DATABASE.INT is defined to encompass the maximum range of integers supported by the DBMS through Ada/SQL.

The DATABASE package also includes the definition of a type SMALLINT, with range corresponding to that supported by the DBMS type SMALLINT through Ada/SQL. The Ada/SQL automated tools will convert Ada integer data types to the corresponding DBMS types as follows: (1) If the Ada type or subtype declaration explicitly references (following a chain of references) INTEGER or DATABASE.INT, then the SQL INTEGER type is used, (2) if the declaration of the Ada type or subtype explicitly references SMALLINT, then the SQL SMALLINT type is used, (3) if none of these types is referenced in the Ada declarations, then the SQL SMALLINT type is used if the range of values is compatible with it, otherwise the SQL INTEGER type is used.

If the range of integers supported by the DBMS is smaller than that supported by Ada, then the Ada/SQL automated tools will issue warning diagnostics upon encountering explicitly declared ranges that extend beyond the capability of the DBMS. The exception NUMERIC_ERROR is raised by the execution of an Ada/SQL operation that would require the DBMS to handle an integer beyond its range.

SQL does not support subtypes, so database operations may be performed without range checking. (An implementation may perform range checking where practical, however, raising CONSTRAINT_ERROR on database operations that would violate subtype constraints.) If range checking is not performed, it is possible for an Ada/SQL statement to cause one or more database columns to contain values outside the ranges defined for those columns. The exception CONSTRAINT_ERROR will be raised, however, when it is attempted to retrieve such values. If the value can be legally stored in the variable used to retrieve it, then the value will be stored before CONSTRAINT_ERROR is raised.

4.7 Special Considerations - LRM section 3.5.6 - real type definition

FUNCTION:

Define the range of values and accuracy for real types

EXAMPLE:

```
type COEFFICIENT is digits 10 range -1.0..1.0; -- a floating point type
type VOLT is delta 0.125 range 0.0..255.0;     -- a fixed point type
  -- taken from LRM sections 3.5.7 and 3.5.9
```

FORMAT:

```
real_type_definition ::= as in Ada, except that it must be static
```

The range and accuracy of real numbers supported by a database management system may not be the same as that supported by the Ada system used to access that DBMS. If the range or accuracy of real numbers supported by the DBMS is smaller than that supported by Ada, then the Ada/SQL automated tools will issue warning diagnostics upon encountering explicitly declared characteristics that extend beyond the capability of the DBMS. The exception NUMERIC_ERROR is raised by the execution of an Ada/SQL operation that would require the DBMS to handle a real number beyond its range. In general, no exception is raised if accuracy is lost as a result of database operations.

The underlying DBMS must support the model numbers (according to the Ada definition) for types that are successfully processed by the automated tools, as well as safe numbers within the ranges of subtypes. The DBMS may also support a wider range of safe numbers.

The comments in the previous section on range checking and CONSTRAINT_ERRORs for INTEGERs are applicable to real numbers as well.

4.7 Special Considerations - LRM section 3.5.7 - floating point constraint

FUNCTION:

Define the range of values and accuracy for floating point types

EXAMPLE:

```
type COEFFICIENT is digits 10 range -1.0..1.0; -- taken from LRM
                                               --  section 3.5.7

type REAL is digits 8;
type MASS is digits 7 range 0.0..1.0E35;

subtype SHORT_COEFF is COEFFICIENT digits 5;
subtype PROBABILITY is REAL range 0.0..1.0;
```

FORMAT:

```
floating_point_constraint ::= as in Ada, except that it must be static
```

The Ada/SQL automated tools will recognize the predefined type FLOAT. Although implementation-specific type names, such as LONG_FLOAT and SHORT_FLOAT, are not recognized, floating point constraints can be used to define types with the corresponding characteristics.

The DATABASE package defines REAL and DOUBLE_PRECISION types, with ranges and accuracies corresponding to those supported by the SQL REAL and DOUBLE PRECISION types as available from the underlying DBMS through Ada/SQL. The Ada/SQL automated tools will convert Ada floating point types to the correspond ing DBMS types as follows: (1) If the Ada type or subtype declaration explicitly references (following a chain of references) DOUBLE_PRECISION, then the SQL DOUBLE PRECISION type is used, (2) if the declaration of the Ada type or subtype explicitly references REAL, then the SQL REAL type is used, (3) if neither DOUBLE_PRECISION nor REAL is referenced in the Ada declarations, then a SQL FLOAT type with appropriate precision is used if the range and accuracy of values is compatible with it, otherwise the SQL DOUBLE_PRECISION type is used. Note that the range and accuracy of the Ada FLOAT type may not correspond to those achievable with the SQL FLOAT type.

The maximum number of floating point digits that can be handled by Ada/SQL through the underlying DBMS is given by the system dependent named number DATABASE.MAX_DIGITS.

For each floating point database type or subtype T, T_NOT_NULL, or T_NOT_NULL_UNIQUE referenced within a schema, the SQL function generator will define named numbers T_SAFE_EMAX, T_SAFE_SMALL, and T_SAFE_LARGE, corresponding to the similarly named attributes of T (or T_NOT_NULL, T_NOT_NULL_UNIQUE, etc.). The attributes of T yield information on the range of exponents supported by the Ada implementation; the corresponding named numbers yield similar information for the DBMS implementation. The DBMS itself may support a wider or a narrower range, which will be accurately reflected in T_SAFE_EMAX. However, T_SAFE_SMALL and T_SAFE_LARGE will not exceed the range of the Ada implementation. As already noted, the DBMS will support all model numbers of a type that is successfully processed by the Ada/SQL automated tools, so the values returned by the DIGITS, MANTISSA, EPSILON, EMAX, SMALL, and LARGE attributes of T are applicable to the DBMS as well as to the Ada implementation. The SQL function generator will also produce the following definitions, corresponding to the machine-dependent attributes of T, except with values applicable to the DBMS implementation:

Type	Generated_Name	Corresponding_Attribute
constant BOOLEAN	T_DATABASE_ROUNDS	T'MACHINE_ROUNDS
constant BOOLEAN	T_DATABASE_OVERFLOWS	T'MACHINE_OVERFLOWS
named number	T_DATABASE_RADIX	T'MACHINE_RADIX
named number	T_DATABASE_MANTISSA	T'MACHINE_MANTISSA
named number	T_DATABASE_EMAX	T'MACHINE_EMAX
named number	T_DATABASE_EMIN	T'MACHINE_EMIN

4.7 Special Considerations - LRM section 3.5.9 - fixed point constraint

FUNCTION:

Define the range of values and accuracy for fixed point types

EXAMPLE:

```
type VOLT is delta 0.125 range 0.0..255.0; -- taken from LRM section
subtype ROUGH_VOLTAGE is VOLT delta 1.0;    -- 3.5.9

DEL : constant := 1.0/2**(WORD_LENGTH - 1);
type FRACTION is delta DEL range -1.0..1.0 - DEL;
```

FORMAT:

```
fixed_point_constraint ::= as in Ada, except that it must be static
```

SQL has no data type directly corresponding to Ada fixed point types. The database representation used for Ada fixed point values is system dependent, depending on the ranges and accuracies desired versus that supported by the various database types, and the efficiencies of processing with different database types. The database representation of any fixed point type must support the model numbers of that type, including the required accuracy of operations. The largest possible number of binary digits in the mantissa of fixed point model numbers that can be handled by Ada/SQL through the underlying DBMS is given by the system dependent named number DATABASE.MAX_MANTISSA. Likewise, the smallest delta that can be supported in a fixed point constraint that has the range constraint -1.0..1.0 is given by DELTA.

For each fixed point database type or subtype T, T_NOT_NULL, or T_NOT_NULL_UNIQUE, referenced within a schema, the SQL function generator will define named numbers T_SAFE_SMALL and T_SAFE_LARGE, corresponding to the similarly named attributes of T (or T_NOT_NULL, T_NOT_NULL_UNIQUE, etc.). The attributes of T yield information on the accuracy and range used by the Ada implementation; the corre-sponding named numbers yield similar information for the DBMS implementation. The DBMS itself may support a greater or lesser accuracy and range (for the underlying base type; it must adequately support the accuracy and range of T), but the named numbers will not exceed the accuracy or range of the Ada implementation. The SQL function generator will also produce the BOOLEAN constants T_DATABASE_ROUNDS and T_DATA_BASE_OVERFLOWS, corresponding to the attributes T'MACHINE_ROUNDS and T'MACHINE_OVERFLOWS, except with values applicable to the DBMS implementation.

4.7 Special Considerations - LRM section 3.6 - array type definition

FUNCTION:

Define array types

EXAMPLE:

```
type VECTOR     is array(INTEGER range <>) of REAL; -- taken from LRM 3.6
type MATRIX     is array(INTEGER range <>, INTEGER range <>) of REAL;
type BIT_VECTOR is array(POSITIVE range <>) of BOOLEAN;
type ROMAN      is array(POSITIVE range <>) of ROMAN_DIGIT;

type TABLE      is array(1..10) of INTEGER;
type SCHEDULE   is array(DAY) of BOOLEAN;
type LINE       is array(1..MAX_LINE_SIZE) of CHARACTER;
```

FORMAT:

```
array_type_definition ::= as in Ada, except that all subtype indications
     and ranges used must either be static or depend on a discriminant.
     Furthermore, the component subtype must be permitted as the type of
     a database column.
```

Two classes of arrays are supported by Ada/SQL:

(1) Arrays with components of a character type (not necessarily the Ada CHARACTER type) that includes only the ASCII characters represented with their usual codes, and with a single index of an integer type. Such arrays are represented within the database as SQL strings, and so are considered scalar by Ada/SQL. The Ada/SQL automated tools will recognize the predefined types CHARACTER and STRING.

(2) All other arrays, which are composite objects in both Ada and Ada/SQL.

Ada/SQL operations available on string arrays correspond to the SQL operations on strings. Operations available on other arrays are the same as those available on private types (see LRM section 3.3.1), except that selection of discriminants is replaced with indexing. Slicing is also available on nonstring arrays with a single index. Indexing and slicing are not available on string arrays, since SQL provides no substring capability.

The schema translator will determine the maximum index range of an array used as a database column for the purpose of allocating database storage as follows:

(1) For a constrained array type/subtype that does not depend on a discriminant, the bounds are given by the index constraint.

(2) An index constraint must be specified (by Ada semantics) for a record component that is of an unconstrained array type. If either bound does not depend on a discriminant, then that value is used as the corresponding database bound. If the lower bound depends on a discriminant of type/subtype D, then the database lower bound is taken as the maximum of D'FIRST and the lower bound of the index type/subtype. If the upper bound depends on a discriminant of type/subtype D, then the database upper bound is taken as the minimum of D'LAST and the upper bound of the index type/subtype.

(3) If a null range is determined from either (1) or (2) above, as appropriate, then the type/subtype of the column must permit null values, because Ada/SQL will only permit nulls to be stored within such a column, which would not be very useful.

Consideration must be given to the index ranges that will be used to allocate database storage for arrays. The Ada/SQL system will determine a mapping of arrays onto the underlying database storage, and larger index ranges may require greater database resources. Excessively large index ranges may not even be supportable by the underlying DBMS, in which case the Ada/SQL automated tools will issue diagnostics. For example, the components in the following declarations could each contain as many as POSITIVE'LAST characters:

```
type PHONE_BOOK(NAME_LENGTH    : POSITIVE := 30;
```

```
                    NUMBER_LENGTH : POSITIVE := 12) is
    record
      NAME   : STRING(1..NAME_LENGTH);
      NUMBER : STRING(1..NUMBER_LENGTH);
    end record;
```

Instead, the definitions should limit the ranges of the discriminants or the possible bounds of the arrays. The hybrid example below shows the discriminant range limited for the NAME component and the array length limited for the NUMBER component:

```
subtype PHONE_NAME_LENGTH is POSITIVE range 1..30;
subtype PHONE_NUMBER_LENGTH is POSITIVE range 1..12;

type PHONE_NUMBER is array (PHONE_NUMBER_LENGTH range <>) of CHARACTER;

type PHONE_BOOK
     (NAME_LENGTH   : PHONE_NAME_LENGTH := PHONE_NAME_LENGTH'LAST;
      NUMBER_LENGTH : POSITIVE := PHONE_NUMBER_LENGTH'LAST) is
  record
    NAME   : STRING(1..NAME_LENGTH);
    NUMBER : PHONE_NUMBER(1..NUMBER_LENGTH);
  end record;
```

4.7 Special Considerations - LRM section 3.7 - record types

FUNCTION:

Record type declarations are used within schemas to declare database tables and also to indicate groupings of fields into subrecords for convenience and uniqueness constraints.

EXAMPLE:

```
type DATE is                          -- taken from LRM section 3.7, not a
  record                              -- database table due to its later
    DAY   : INTEGER range 1..31;  -- use as a component of another
    MONTH : MONTH_NAME;           -- record type
    YEAR  : INTEGER range 0..4000;
  end record;

type STOCK_TRANSACTION is            -- a database table that includes
  record                             -- the  above subrecord
    TRADE_DATE  : DATE;              -- component types are assumed to be
    ACCOUNT     : ACCOUNT_NUMBER; --  defined appropriately
    COMPANY     : COMPANY_NAME;
    SHARES      : SHARE_COUNT;
    TRANSACTION : BOUGHT_OR_SOLD;
    PRICE       : STOCK_PRICE;
  end record;
```

FORMAT:

```
record_type_definition ::= as in Ada, except that types/subtypes and
     expressions  must either be static or depend on a discriminant of
     the record. Record components must be of types/subtypes that are
     legal for database columns.
```

The declaration of a record type also declares a database table, of the same name as the record type, if:

(1) It occurs within a schema package, and

(2) The record type name is not used as the type mark of a component of another record declared within the same schema package. For the purpose of this determination, a subtype name differing only in the use of the _NOT_NULL or _NOT_NULL_UNIQUE suffixes is considered to be the same name.

Each component of a record declaring a database table defines a column within that table. The column names are the same as the component names, and the SQL types of the columns are as discussed with each class of type declaration. Columns defined as record or array types are composite columns, with the underlying database columns being the non-composite subcomponents of the types. Composite columns may be given the constraints _NOT_NULL or _NOT_NULL_UNIQUE only if all subcomponents are constrained to prohibit null values. Within the data manipulation language, composite columns may be used wherever private types may be used, as discussed for LRM section 3.3.1, except that discriminant selection should be replaced with selection of any component for record columns and indexing for array columns.

There is no way to indicate whether subcolumns not depending on a discriminant are null when referencing composite columns, although the entire composite column may be specified as null by using an indicator variable. When writing to the database or specifying values for comparison, such subcolumns are taken to be non-null unless the entire composite column is null. A null value denoted by an indicator variable causes all subcolumns, including those which would otherwise be non-null based on discriminant values, to be considered null. When reading from the database, the indicator variable is set to indicate null if and only if all subcolumns are null. If any subcolumns not depending on a discriminant are null in a composite column that is not completely null, then the NULL_ERROR exception will be raised as if a null value had been retrieved without an indicator variable specified (see comments on dpANS section 6.6). For program variables of a composite type used to retrieve composite columns, component values not depending on discriminants are left undefined if the corresponding subcolumn contains a null value. If the individual null status of subcolumns not depending on discriminants is important, then all subcolumns should be referenced individually.

When adding rows to a table with INSERT_INTO statements, it is not necessary to specify values for all columns. SQL uses null values for unspecified columns, but Ada/SQL uses the default expressions given for the record components used to define the table, if such defaults are provided. Nested levels of subrecords may give rise to conflicting default expressions: the default values used are those that would be used by Ada for an object of the record type whose declaration also declared the database table to which rows are being added. In short, outer defaults take precedence over inner defaults. When inserting a new row into a table, null values are used for all columns without explicit or default values. It is an SQL error, returned in the standard fashion of the data manipulation language, to attempt to insert a null value into a column not permitting null values. In other words, INSERT_INTO statements must specify values for all non-null columns of the table in question, either by explicit reference or by default expressions obtained from the declarations of the applicable record types.

4.7 Special Considerations - LRM section 3.7.: - discriminants

FUNCTION:

As in Ada, define subrecords containing variable-length arrays. Also define variant parts of records.

EXAMPLE:

```
type SHIP_NAME_LENGTH is range 1..20;

type SHIP_NAME_STRING is new STRING;

type SHIP_NAME(LENGTH : SHIP_NAME_LENGTH := SHIP_NAME_LENGTH'LAST) is
  record
    NAME : SHIP_NAME_STRING(1..LENGTH);
  end record;
```

FORMAT:

```
discriminant_part ::= as in Ada, except that all type marks and default
     expressions used must either be static or depend on a discriminant
```

Null values are not permitted in discriminant columns, regardless of the type/subtype name of the discriminant. In keeping with Ada philosophy, a database discriminant value may be modified only by updating all columns defined by the record type containing the discriminant. Discriminants may also be used to choose variant parts within records. When updating discriminant values, values need not and may not be specified for variant parts not chosen by the discriminant values.

4.7 Special Considerations - LRM section 8.5 - renaming declaration

FUNCTION:

Provide alternate names for entities.

EXAMPLE:

```
MAXLEN : INTEGER renames SHIP_NAME_MAXIMUM_LENGTH;
                                    -- renaming an object
package SHIPS renames SHIP_DATA_DEFINITION_PACKAGE;
                                    -- renaming a package
```

FORMAT:

```
renaming_declaration ::= as in Ada, except that only constants and
    packages may be renamed within schema packages
```

Renaming declarations permitted within schema packages are restricted to those that rename entities that may be declared or referenced within schemas. Hence, only constants and packages may be renamed.

4.7 Special Considerations - LRM section 13.1 - representation clause

FUNCTION:

Within schemas, specify how enumeration types are to be represented.

EXAMPLE:

```
type CLASSIFICATION is (UNCLASSIFIED, CONFIDENTIAL, SECRET,
 TOP_SECRET);

for CLASSIFICATION use (1,2,4,8); -- an enumeration representation
                                   -- clause
```

FORMAT:

```
representation_clause ::= as in Ada
```

All types of representation clauses may be used within schemas, as they apply to the types and objects that may be declared. The Ada/SQL automated tools will ignore all representation clauses other than enumeration representation clauses, however. Enumeration representation clauses will, if possible, have the effect of changing the database representation for enumeration types.

5.1 - 5.3 Module Language

There are no corresponding sections within the Ada/SQL standard. Ada/SQL uses an embedded language, and therefore does not require the module language.

6.1 <close statement>

FUNCTION:

Close a cursor.

EXAMPLE:

```
CURSOR : CURSOR_NAME;
. . .
CLOSE(CURSOR);
```

FORMAT:

```
<close statement> ::=
     CLOSE ( <cursor name> ) ;
```

The <close statement> is an Ada procedure.

```
<cursor name> ::= a program variable of type CURSOR_NAME,  which is
     a private type defined by the implementation
```

6.2 <commit statement>

FUNCTION:

Terminate the current transaction with commit.

EXAMPLE:

```
COMMIT_WORK;
```

FORMAT:

```
<commit statement> ::=
     COMMIT_WORK ;
```

The <commit statement> is an Ada procedure.

6.3 <declare cursor>

FUNCTION:

Define a cursor.

EXAMPLE:

```
CURSOR : CURSOR_NAME;

package E  is new ANALYST_CORRELATION_NAME; -- employees \ see section
package M is new ANALYST_CORRELATION_NAME;  -- managers  / 3.20
. . .

DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC      ( NAME & SALARY & MANAGER,
  FROM      => ANALYST ),
  ORDER_BY => MANAGER & DESC(SALARY) );

  -- variations: ORDER_BY => 3 & DESC(2)
  --             ORDER_BY => ASC(3) & DESC(SALARY), etc.

DECLAR ( CURSOR , CURSOR_FOR =>
      SELEC  ( NAME & SALARY & MANAGER,
      FROM  => ANALYST,
      WHERE => SALARY > 25_000.00 )
  & UNION (
      SELEC  ( E.NAME & E.SALARY & E.MANAGER,
      FROM  => E.ANALYST & M.ANALYST, -- see section 3.20
      WHERE => EQ ( E.MANAGER , M.NAME )
      AND            E.SALARY > M.SALARY ) ) );

    -- variations: UNION_ALL
```

FORMAT:

```
<declare cursor> ::=
     DECLAR ( <cursor name> , CURSOR_FOR =>
       <cursor specification> ) ;
```

The <declare cursor> statement is an Ada procedure named DECLAR (the SQL keyword DECLARE is an Ada reserved word). Its first parameter is the cursor to be declared. Its second parameter is named CURSOR_FOR to retain the SQL keyword, and is a specification of the retrieval to be performed by the cursor. The third parameter (discussed below as part of the <cursor specification>) is named ORDER_BY to handle the <order by clause>.

```
<cursor name> ::= see section 6.1

<cursor specification> ::=
     <query expression> [ ,
     <order by clause> ]
```

The <order by clause> is optional. Since it is the third parameter to DECLAR, a comma must separate it from the <query expression>.

```
<query expression> ::=
          <query term>
       |  <query expression> & { UNION  |  UNION_ALL } ( <query term> )
       |  <query expression> & { UNION  |  UNION_ALL }   <query term>
```

Since UNION and UNION_ALL cannot be made infix operators, the ampersand is used to connect the two items being UNIONed. UNION and UNION_ALL are functions on <query term>s that are used to keep the SQL keywords in the operation and that return an indication of whether or not the ALL option was used. Parentheses are required around UNION and UNION_ALL's parameter by Ada syntax: they are shown as optional here because they may be supplied from the next

production, which is used within SQL to show precedence of UNIONs.

```
<query term> ::=
     <query specification>  |  ( <query expression> )

<query specification> ::= see section 3.25.

<order by clause> ::=
     ORDER_BY => <sort specification> [ { & <sort specification> } ... ]
```

As noted above, the third parameter to DECLAR is named ORDER_BY. The <sort specification>s cannot be joined by commas in Ada; so ampersands are used.

```
<sort specification> ::=
                    <sort column specification>
       |   ASC  ( <sort column specification> )
       |   DESC ( <sort column specification> )
```

Ascending (default) and descending sorts cannot be indicated by appending the ASC or DESC keyword to the column indicator; ASC and DESC are instead made Ada functions.

```
<sort column specification> ::=
     <column number>   |   <column specification>

<column number> ::= a positive integer of type COLUMN_NUMBER
```

Ada's typing is used to define a <column number>.

```
<column specification> ::= see section 3.7
```

6.4 <delete statement: positioned>

FUNCTION:

Delete a row of a table based on the current position of a cursor.

EXAMPLE:

```
CURSOR : CURSOR_NAME;
. . .
DELETE_FROM ( ANALYST,
WHERE_CURRENT_OF => CURSOR );
```

FORMAT:

```
<delete statement: positioned> ::=
     DELETE_FROM ( <table name> ,
     WHERE_CURRENT_OF => <cursor name> ) ;
```

The <delete statement: positioned> is an Ada procedure.

```
<table name> ::= see section 3.20

<cursor name> ::= see section 6.1
```

6.5 <delete statement: searched>

FUNCTION:

Delete rows of a table based on a search criterion.

EXAMPLE:

```
DELETE_FROM ( ANALYST,
WHERE       => SALARY > 25_000.00 );

DELETE_FROM ( ANALYST );
```

FORMAT:

```
<delete statement: searched> ::=
     DELETE_FROM ( <table name> [ ,
     WHERE       => <search condition> ] ) ;
```

The <delete statement: searched> is an Ada procedure. The WHERE keyword names the second parameter. The WHERE parameter may be omitted.

```
<table name> ::= see section 3.20

<search condition> ::= see section 3.18
```

6.6 <fetch statement>

FUNCTION:

Position a cursor on the next row of a table and assign values in that row to program variables.

EXAMPLE:

```
CURRENT_EMPLOYEE        : ANALYST_NAME;
HIS_SALARY              : ANALYST_SALARY;
HIS_MANAGER             : MANAGER_NAME;
CURSOR                  : CURSOR_NAME;
LAST                    : NATURAL;
IND_VAR                 : INDICATOR_VARIABLE;
 . . .
DECLAR ( CURSOR , CURSOR_FOR =>
  SELEC ( NAME & SALARY & MANAGER,
  FROM => ANALYST ) );
FETCH ( CURSOR );
INTO ( CURRENT_EMPLOYEE , LAST );
INTO ( HIS_SALARY , IND_VAR );        -- variation: INTO ( HIS_SALARY );
INTO ( HIS_MANAGER, LAST, IND_VAR );
```

FORMAT:

```
<fetch statement> ::=
     FETCH ( <cursor name> ) ;
     INTO ( <result specification> [ , <cursor name> ] ) ;
 [ { INTO ( <result specification> [ , <cursor name> ] ) ; } ... ]
```

It is not possible to string the result variables together as with SQL. Consequently, a FETCH procedure call is followed by as many calls to INTO as are required to retrieve the values of each column in the row. Each INTO returns one column value. The NOT_FOUND_ERROR exception will be raised if a FETCH is performed on a cursor for which all rows (if any) have already been returned. If several tasks within the same program are simultaneously performing database retrievals, the <cursor name> used in the FETCH must be specified as the final parameter to INTO procedures for that FETCH. If simultaneous database retrievals are not being performed, the <cursor name> parameter may be omitted from the INTO calls.

```
<cursor name> ::= see section 6.1

<result specification> ::=
    <result program variable>
      [ , <last variable> ] [ , <indicator variable> ]

<result program variable> ::= program variable to obtain column value
     from database, which must be of a type comparable with that of the
     database column being retrieved.  Program variables within <result
     specification>s may also be expressed as type conversions, as would
     be legal for any Ada out actual parameter.

<last variable> ::= program variable to obtain the value of the last
     index position used in retrieving array values. Used when and only
     when <result program variable> is of type array. For one dimension-
     al arrays, <last variable> is of the same type as the array index.
     For arrays of higher dimensionality, <last variable> is a record
     with components corresponding to each array index type. The record
     type, named A_LAST, where A, A_NOT_NULL, or A_NOT_NULL_UNIQUE is
     the name of the array type, is defined by the SQL function genera-
     tor for all schema packages containing a database column or subcol-
     umn of the array type. Components of the record are named LAST_1,
     LAST_2, etc.
```

<indicator variable> ::= optional program variable of type INDICATOR_-VARIABLE, set to NULL_VALUE if the database column retrieved contains a null value, else set to NOT_NULL. If a null value is retrieved from the database but no <indicator variable> is specified, the NULL_ERROR exception will be raised.

6.7 <insert statement>

FUNCTION:

Create new rows in a table.

EXAMPLE:

```
NEW_EMPLOYEE  : ANALYST_NAME;
HIS_SALARY    : ANALYST_SALARY;
HIS_MANAGER   : MANAGER_NAME;
. . .
INSERT_INTO ( ANALYST ( EMPLOYEE & SALARY & MANAGER ) ,
  VALUES <= NEW_EMPLOYEE and HIS_SALARY and HIS_MANAGER );

INSERT_INTO ( ANALYST ,
  SELEC ( '*',
  FROM => NEW_ANALYST_FILE ) );

-- assume NEW_ANALYST_FILE is another database table structured
-- identically to the ANALYST table
```

FORMAT:

```
<insert statement> ::=
     INSERT_INTO ( <table name> [ ( <insert column list> ) ] ,
   { VALUES <= <insert value list> }   <query specification> ) ;
```

The INSERT INTO statement is a function with two parameters. The first parameter indicates the table and (optionally) columns to be affected. If the <insert column list> is specified, then the <table name> is expressed as a function with one parameter (the <insert column list>). The parentheses around the Ada function parameter exactly match the SQL parentheses. If the <insert column list> is not specified, then the <table name> is a function with no parameters. A comma separates the first and second argument to INSERT_INTO. The second argument can be either an explicit list of values or a <query specification>. For the explicit list of values, the SQL keyword VALUES is followed by an Ada/SQL "gets" operator (<=), then the list of values. The SQL parenthesis notation is not used. A <query specification> is indicated by a call to the SELEC function (see section 3.25).

```
<table name> ::= see section 3.20.  If an <insert column list> is used
     and the <table name> includes an <authorization identifier>, then
     the syntax for the <table name> is <authorization identifier>-
     <table identifier>.  This is one of the three contexts within
     Ada/SQL where <table name> syntax is not the usual <authorization
     identifier>.<table identifier>.

<insert column list> ::=
     <column name> [ { & <column name> } ... ]
```

SQL uses commas to separate the column names, Ada/SQL uses ampersands instead.

```
<column name> ::= an unqualified column name

<insert value list> ::=
     <insert value> [ { and <insert value> } ... ]
```

SQL uses commas to separate the insert values, Ada/SQL uses an overloaded "and" operator.

```
<insert value> ::=
     <value specification>  |  NULL_VALUE
```

SQL uses NULL, which is an Ada reserved word.

```
<value specification> ::= see section 3.6

<query specification> ::= see section 3.25
```

6.8 <open statement>

FUNCTION:

Open a cursor.

EXAMPLE:

```
CURSOR : CURSOR_NAME;
. . .
OPEN(CURSOR);
```

FORMAT:

```
<open statement> ::=
    OPEN ( <cursor name> ) ;
```

The <open statement> is an Ada procedure.

```
<cursor name> ::= a program variable of type CURSOR_NAME
```

Note that SQL evaluates program variables used to declare a cursor when the cursor is opened. In Ada/SQL, these program variables are evaluated when the cursor is declared.

6.9 <rollback statement>

FUNCTION:

Terminate the current transaction with rollback.

EXAMPLE:

```
ROLLBACK_WORK;
```

FORMAT:

```
<rollback statement> ::=
     ROLLBACK_WORK ;
```

The <rollback statement> is an Ada procedure.

6.10 <select statement>

FUNCTION:

Specify a table and assign the values in the single row of that table to program variables.

EXAMPLE:

```
DESIRED_EMPLOYEE : ANALYST_NAME;
HIS_SALARY       : ANALYST_SALARY;
HIS_MANAGER      : MANAGER_NAME;
LAST             : NATURAL;
. . .
SELEC  ( SALARY & MANAGER ,          -- variations: SELECT_ALL,
FROM  => ANALYST,                    --             SELECT_DISTINCT
WHERE => EQ(NAME,DESIRED_EMPLOYEE) );
INTO(HIS_SALARY);
INTO(HIS_MANAGER,LAST);
```

FORMAT:

```
<select statement> ::=
   [ SELEC  |  SELECT_ALL  |  SELECT_DISTINCT ] ( <select list> ,
     <table expression> ) ;
     INTO ( <result specification> ) ;
  [ { INTO ( <result specification> ) ; } ... ]
```

The SELECT INTO statement is an Ada procedure. It is not possible to specify the ALL or DISTINCT keywords separately, so they are part of the procedure name if used. The name of the procedure is SELEC if neither keyword is used, since SELECT is an Ada reserved word. The SELECT INTO procedures have three parameters which must, of course, be surrounded by parentheses and separated by commas. The first is the <select list>, as described in section 3.25. The second and third parameters are the FROM and WHERE clauses from the <table expression>, and so are named FROM and WHERE, respectively. (SQL does not permit GROUP BY and HAVING clauses in <select statement>s. <table expression>s are discussed in section 3.19; when used within <select statement>s the optional <group by clause> and <having clause> must be omitted.)

The INTO keyword cannot be embedded within the SELECT INTO statement in Ada, nor can result variables be listed together, separated by commas. Consequently, one INTO call for each column retrieved is made following the SELECT INTO statement, with the exact same format and meaning as described in section 6.6 for the FETCH statement, except that a <cursor name> may not be specified. Consequently, tasks within a program must not perform more than one simultaneous <select statement>. If multiple retrievals must be performed simultaneously, FETCH statements must be used to avoid erroneous results. The NOT_FOUND_ERROR exception will be raised if the SELECT INTO statement retrieved no rows, and UNIQUE_ERROR will be raised if it retrieved more than one. The values returned by the INTO statements are undefined when called directly following errors.

```
<select list> ::= see section 3.25

<table expression> ::= see section 3.19

<result specification> ::= see section 6.6
```

6.11 <update statement: positioned>

FUNCTION:

Modify a row of a table based on a cursor's current position.

EXAMPLE:

```
CURSOR : CURSOR_NAME;
. . .
UPDATE ( ANALYST ,
SET    => SALARY  <= 2.0*SALARY
     and MANAGER <= NULL_VALUE ,
WHERE_CURRENT_OF => CURSOR );
```

FORMAT:

```
<update statement: positioned> ::=
     UPDATE ( <table name> ,
     SET    => <set clause> [ { and <set clause> } ... ] ,
     WHERE_CURRENT_OF => <cursor name> ) ;
```

The Ada UPDATE procedure has three parameters: the name of the table to be updated, a list of columns and values to be set, and the cursor used to determine the current row. Named associations on the last two parameters serve to get the SQL keywords into the Ada program. The <set clause>s are separated by "and"s in Ada/SQL, instead of commas as in SQL.

```
<table name> ::= see section 3.20

<set clause> ::=
     <object column> <= { <value expression>   NULL_VALUE }

<object column> ::= <column name>

<column name> ::= an unqualified column name

<value expression> ::= see section 3 9
```

The = operator cannot be overloaded for use in a <set clause>, so Ada/SQL uses <= instead, to be indicative of the direction of the assignment. NULL_VALUE is used instead of NULL, which is an Ada reserved word.

```
<cursor name> ::= see section 6.1
```

6.12 <update statement: searched>

FUNCTION:

Update rows of a table based on a search condition.

EXAMPLE:

```
GENEROUS_BOSS : MANAGER_NAME;
. . .
UPDATE ( ANALYST ,
SET   => SALARY <= 1.2*SALARY ,
WHERE => EQ(MANAGER,GENEROUS_BOSS) );

UPDATE ( ANALYST,
SET   => SALARY <= 1.05*SALARY ); -- cost of living raise
```

FORMAT:

```
<update statement: searched> ::=
    UPDATE ( <table name> ,
    SET   => <set clause> [ { and <set clause> } ... ] [ ,
    WHERE => <search condition> ] ) ;
```

This version of the Ada UPDATE procedure has three parameters: the name of the table to be updated, a list of columns and values to be set, and the search condition used to determine which rows are to be updated. The search condition is optional. Named associations on the last two parameters serve to get the SQL keywords into the Ada program. The <set clause>s are separated by "and"s instead of by commas as in SQL.

```
<table name> ::= see section 3.20

<set clause> ::= see section 6.11

<search condition> ::= see section 3.18
```

I.6. Appendix A. <embedded SQL host program>

The definition is extended to include

```
<embedded SQL host program> ::= <embedded SQL Ada program>
```

An <embedded SQL Ada program> consists of an Ada program which includes Ada/SQL data manipulation statements as defined in Section 6. No other special embedded notation (such as EXEC SQL, etc., used with other languages) is required, since an <embedded SQL Ada program> conforms to precise Ada syntax and may be directly processed by the Ada compiler rather than being precompiled. This also means that program variables are used directly within Ada/SQL statements, without the leading colon:

```
<embedded variable name> ::= <host identifier>
```

Program variables may be of user-defined data types as discussed in section 2.2. Operations on database and program values must be between comparable data types, as defined in section 3.9.

I.7. Appendix B. <embedded exception declaration>

Error conditions in Ada/SQL always raise exceptions, that are then processed in the normal Ada manner. Exceptions may not be "turned off", nor may program control be transferred in any manner other than the standard Ada exception handlers.

Appendix II
Prototype Implementation Software

II.1. Introduction

This section of the report will list code developed to implement a prototype of the Ada/SQL DDL and DML. A brief description of the input. output. and source file listings is included before both sections. All Ada code is listed in order of compilation. All of this code is located on-line in the directory [CONTR17.ADASQL.DDL] and [CONTR17.ADASQL.DDL] on the (Arpanet accessible) NOSC-TECR VAX computer system.

II.2. Ada/SQL DDL Prototype Implementation Software

This section contains a program illustrating the translation of the Ada DDL to the DDL required by other systems. It is discussed in Section 2.0 of the main body of this report.

Input file: BOATS.ADA - an example of our Ada DDL, also a legal Ada package. This file is located in the appendix at the end of this section.

Output file: DDL.OUT - script of screen output when the program was run using the Data General (Rolm) validated Ada compiler. Shows (1) echo and pretty print of input, (2) simple DDL currently required by the prototype database management system used to demonstrate the DML (see [CONTR17.ADASQL.DML]), and (3) DDL required by DAMES as interfaced with Ada.

Notes on the main program: CREATE_STREAM, SET_STREAM, OPEN_INPUT and CLOSE_INPUT are used to manage the input file. CREATE_LINE and SET_LINE manage the output file. SCAN_DDL reads the Ada DDL file as input, builds data structures representative of the input, and returns the name of the package read. DISPLAY_DDL uses the data structures to pretty print the input and to print a tree of subtypes. GENERATE_SIMPLE_DDL uses the data structures to translate the input into our simple DDL, while GENERATE_DAMES_DDL performs the same function for the DAMES DDL.

Source files (.ADA) in compilation order:

```
Filename  Package name     Description
--------  ------------     -----------
TOKEN     TOKEN_INPUT      Manage input file, simplify reading input as tokens

TXTPRT    TEXT_PRINT       Manage output file, including continuation lines
                           Also provide minimum width and default format
                           printing of numbers
                           Same as TEXT_PRINT for DML, except added phantoms

DDLDEFS   DDL_DEFINITIONS  Defines data structures built by SCAN_DDL

LISTUTIL  LIST_UTILITIES   Handy routines for manipulating data structures

READDDL   READ_DDL         Read Ada DDL input, building data structures

SHOWDDL   SHOW_DDL         Pretty print DDL input, print subtype tree

SIMDDL    SIMPLE_DDL       Generate simple DDL from data structures

DAMESDDL  DAMES_DDL        Generate DAMES DDL from data structures

MAIN      MAIN             Main program
```

```
with TEXT_IO;
  use TEXT_IO;

package TOKEN_INPUT is

  type INPUT_STREAM is private;

  package INTEGER_IO is new TEXT_IO.INTEGER_IO(INTEGER);
    use INTEGER_IO;

  function CREATE_STREAM(CARD_LENGTH : POSITIVE) return INPUT_STREAM;

  procedure SET_STREAM(STREAM : INPUT_STREAM);

  procedure OPEN_INPUT(STREAM : INPUT_STREAM;
                       NAME   : STRING);

  procedure OPEN_INPUT(NAME : STRING);

  procedure CLOSE_INPUT(STREAM : INPUT_STREAM);

  procedure CLOSE_INPUT;

  procedure GET_STRING(STREAM : in  INPUT_STREAM;
                       STR    : out STRING;
                       LAST   : out NATURAL);

  procedure GET_STRING(STR  : out STRING;
                       LAST : out NATURAL);

  function GET_INTEGER(STREAM : INPUT_STREAM) return INTEGER;

  function GET_INTEGER return INTEGER;

  procedure GOBBLE(STREAM : INPUT_STREAM;
                   STR    : STRING);

  procedure GOBBLE(STR : STRING);

private

  type INPUT_RECORD(CARD_LENGTH : POSITIVE) is
    record
      BUFFER : STRING(1..CARD_LENGTH);
      FILE   : FILE_TYPE;
      NEXT   : POSITIVE := 1;
      LAST   : NATURAL := 0;
    end record;

  type INPUT_STREAM is access INPUT_RECORD;

end TOKEN_INPUT;

package body TOKEN_INPUT is

  DEFAULT_STREAM : INPUT_STREAM;
```

```
function CREATE_STREAM(CARD_LENGTH : POSITIVE) return INPUT_STREAM is
begin
  return new INPUT_RECORD(CARD_LENGTH);
end CREATE_STREAM;

procedure SET_STREAM(STREAM : INPUT_STREAM) is
begin
  DEFAULT_STREAM := STREAM;
end SET_STREAM;

procedure OPEN_INPUT(STREAM : INPUT_STREAM;
                     NAME   : STRING) is
begin
  OPEN(STREAM.FILE,IN_FILE,NAME);
end OPEN_INPUT;

procedure OPEN_INPUT(NAME : STRING) is
begin
  OPEN_INPUT(DEFAULT_STREAM,NAME);
end OPEN_INPUT;

procedure CLOSE_INPUT(STREAM : INPUT_STREAM) is
begin
  CLOSE(STREAM.FILE);
end CLOSE_INPUT;

procedure CLOSE_INPUT is
begin
  CLOSE_INPUT(DEFAULT_STREAM);
end CLOSE_INPUT;

function ALPHABETIC(C : CHARACTER) return BOOLEAN is
begin
  return C in 'A'..'Z' or else C in 'a'..'z' or else C = '_';
end ALPHABETIC;

function NUMERIC(C : CHARACTER) return BOOLEAN is
begin
  return C in '0'..'9' or else C = '_';
end NUMERIC;

function WHITESPACE(C : CHARACTER) return BOOLEAN is
begin
  return C = ' ' or else C = ASCII.HT;
end WHITESPACE;

procedure NEXT_LINE(STREAM : INPUT_STREAM) is
begin
  loop
    GET_LINE(STREAM.FILE,STREAM.BUFFER,STREAM.LAST);
    exit when STREAM.LAST >= 2 and then STREAM.BUFFER(1..2) /= "--";
    exit when STREAM.LAST = 1;
  end loop;
  STREAM.NEXT := 1;
end NEXT_LINE;

procedure NEXT_TOKEN(STREAM : INPUT_STREAM) is
```

```
begin
  loop
    if STREAM.NEXT > STREAM.LAST then
      NEXT_LINE(STREAM);
    end if;
    if STREAM.BUFFER(STREAM.NEXT) = '-' and then
        STREAM.NEXT < STREAM.LAST and then
        STREAM.BUFFER(STREAM.NEXT+1) = '-' then
      NEXT_LINE(STREAM);
    end if;
    exit when not WHITESPACE(STREAM.BUFFER(STREAM.NEXT));
    STREAM.NEXT := STREAM.NEXT + 1;
  end loop;
end NEXT_TOKEN;

function TOKEN_END(STREAM : INPUT_STREAM) return POSITIVE is
  C   : CHARACTER;
  PTR : POSITIVE;
begin
  NEXT_TOKEN(STREAM);
  PTR := STREAM.NEXT;
  while PTR <= STREAM.LAST loop
    C := STREAM.BUFFER(PTR);
    exit when WHITESPACE(C);
    case STREAM.BUFFER(STREAM.NEXT) is
      when 'A'..'Z' | 'a'..'z' =>
        exit when not ALPHABETIC(C) and then not NUMERIC(C);
      when '0'..'9' | '-' | '+' =>
        exit when not NUMERIC(C);
      when others =>
        exit when ALPHABETIC(C) or else NUMERIC(C);
    end case;
    PTR := PTR + 1;
  end loop;
  return PTR - 1;
end TOKEN_END;

procedure GET_STRING(STREAM : in  INPUT_STREAM;
                     STR    : out STRING;
                     LAST   : out NATURAL) is
  TOKEND,
  TLAST : POSITIVE;
begin
  TOKEND := TOKEN_END(STREAM);
  TLAST := STR'FIRST + TOKEND - STREAM.NEXT;
  STR(STR'FIRST..TLAST) := STREAM.BUFFER(STREAM.NEXT..TOKEND);
  LAST := TLAST;
  STREAM.NEXT := TOKEND + 1;
end GET_STRING;

procedure GET_STRING(STR  : out STRING;
                     LAST : out NATURAL) is
begin
  GET_STRING(DEFAULT_STREAM, STR, LAST);
end GET_STRING;

function GET_INTEGER(STREAM : INPUT_STREAM) return INTEGER is
```

```
    TOKEND  : POSITIVE;
    INT,
    LAST    : INTEGER;
  begin
    TOKEND := TOKEN_END(STREAM);
    GET(STREAM.BUFFER(STREAM.NEXT..TOKEND),INT,LAST);
    STREAM.NEXT := TOKEND + 1;
    return INT;
  end GET_INTEGER;

  function GET_INTEGER return INTEGER is
  begin
    return GET_INTEGER(DEFAULT_STREAM);
  end GET_INTEGER;

  procedure GOBBLE(STREAM : INPUT_STREAM;
                   STR    : STRING) is
    S    : STRING(1..STREAM.CARD_LENGTH);
    LAST : INTEGER;
  begin
    GET_STRING(STREAM,S,LAST);
    if S(1..LAST) /= STR then
      raise CONSTRAINT_ERROR;
    end if;
  end GOBBLE;

  procedure GOBBLE(STR : STRING) is
  begin
    GOBBLE(DEFAULT_STREAM,STR);
  end GOBBLE;

end TOKEN_INPUT;
```

```
with TEXT_IO;
  use TEXT_IO;

package TEXT_PRINT is

  type LINE_TYPE is limited private;

  type BREAK_TYPE is (BREAK, NO_BREAK);

  type PHANTOM_TYPE is private;

  procedure CREATE_LINE(LINE : in out LINE_TYPE; LENGTH : in POSITIVE);

  procedure SET_LINE(LINE : in LINE_TYPE);

  function CURRENT_LINE return LINE_TYPE;

  procedure SET_INDENT(LINE   : in LINE_TYPE; INDENT : in NATURAL);
  procedure SET_INDENT(INDENT : in NATURAL);

  procedure SET_CONTINUATION_INDENT(LINE   : in LINE_TYPE;
                                    INDENT : in INTEGER);
  procedure SET_CONTINUATION_INDENT(INDENT : in INTEGER);

  function MAKE_PHANTOM(S : STRING) return PHANTOM_TYPE;

  procedure SET_PHANTOMS(LINE          : in LINE_TYPE;
                         START_PHANTOM,
                         END_PHANTOM   : in PHANTOM_TYPE);

  procedure SET_PHANTOMS(START_PHANTOM, END_PHANTOM : in PHANTOM_TYPE);

  procedure PRINT(FILE : in FILE_TYPE;
                  LINE : in LINE_TYPE;
                  ITEM : in STRING;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(FILE : in FILE_TYPE;
                  ITEM : in STRING;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(LINE : in LINE_TYPE;
                  ITEM : in STRING;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(ITEM : in STRING;
                  BRK  : in BREAK_TYPE := BREAK);

  procedure PRINT_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE);
  procedure PRINT_LINE(FILE : in FILE_TYPE);
  procedure PRINT_LINE(LINE : in LINE_TYPE);
  procedure PRINT_LINE;

  procedure BLANK_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE);
  procedure BLANK_LINE(FILE : in FILE_TYPE);
  procedure BLANK_LINE(LINE : in LINE_TYPE);
  procedure BLANK_LINE;

  generic
    type NUM is range <>;
```

```
package INTEGER_PRINT is

  procedure PRINT(FILE : in FILE_TYPE;
                  LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);

  procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM);

end INTEGER_PRINT;

generic
  type NUM is digits <>;
package FLOAT_PRINT is

  procedure PRINT(FILE : in FILE_TYPE;
                  LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);

  procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM);

end FLOAT_PRINT;

NULL_PHANTOM : constant PHANTOM_TYPE;

LAYOUT_ERROR : exception renames TEXT_IO.LAYOUT_ERROR;

private

type PHANTOM_TYPE is access STRING;

type LINE_REC(LENGTH : INTEGER) is
  record
    USED_YET            : BOOLEAN := FALSE;
    INDENT              : INTEGER := 0;
    CONTINUATION_INDENT : INTEGER := 2;
    BREAK               : INTEGER := 1;
    INDEX               : INTEGER := 1;
    DATA                : STRING(1..LENGTH);
    START_PHANTOM,
```

```
      END_PHANTOM          : PHANTOM_TYPE := NULL_PHANTOM;
    end record;

  type LINE_TYPE is access LINE_REC;

  NULL_PHANTOM : constant PHANTOM_TYPE := new STRING'("");

end TEXT_PRINT;

package body TEXT_PRINT is

  DEFAULT_LINE : LINE_TYPE;

  procedure CREATE_LINE(LINE : in out LINE_TYPE; LENGTH : in POSITIVE) is
  begin
    LINE := new LINE_REC(LENGTH);
  end CREATE_LINE;

  procedure SET_LINE(LINE : in LINE_TYPE) is
  begin
    DEFAULT_LINE := LINE;
  end SET_LINE;

  function CURRENT_LINE return LINE_TYPE is
  begin
    return DEFAULT_LINE;
  end CURRENT_LINE;

  procedure SET_INDENT(LINE   : in LINE_TYPE; INDENT : in NATURAL) is
  begin
    if INDENT >= LINE.LENGTH then
      raise LAYOUT_ERROR;
    end if;
    if LINE.INDEX = LINE.INDENT + 1 then
      for I in 1..INDENT loop
        LINE.DATA(I) := ' ';
      end loop;
      LINE.INDEX := INDENT + 1;
    end if;
    LINE.INDENT := INDENT;
  end SET_INDENT;

  procedure SET_INDENT(INDENT : in NATURAL) is
  begin
    SET_INDENT(DEFAULT_LINE, INDENT);
  end SET_INDENT;

  procedure SET_CONTINUATION_INDENT(LINE   : in LINE_TYPE;
                                    INDENT : in INTEGER) is
  begin
    if LINE.INDENT + INDENT >= LINE.LENGTH or else LINE.INDENT + INDENT < 0
        then
      raise LAYOUT_ERROR;
    end if;
    LINE.CONTINUATION_INDENT := INDENT;
  end SET_CONTINUATION_INDENT;
```

```
procedure SET_CONTINUATION_INDENT(INDENT : in INTEGER) is
begin
  SET_CONTINUATION_INDENT(DEFAULT_LINE, INDENT);
end SET_CONTINUATION_INDENT;

function MAKE_PHANTOM(S : STRING) return PHANTOM_TYPE is
begin
  return new STRING'(S);
end MAKE_PHANTOM;

procedure SET_PHANTOMS(LINE          : in LINE_TYPE;
                       START_PHANTOM,
                       END_PHANTOM   : in PHANTOM_TYPE) is
begin
  LINE.START_PHANTOM := START_PHANTOM;
  LINE.END_PHANTOM := END_PHANTOM;
end SET_PHANTOMS;

procedure SET_PHANTOMS(START_PHANTOM, END_PHANTOM : in PHANTOM_TYPE) is
begin
  SET_PHANTOMS(DEFAULT_LINE, START_PHANTOM, END_PHANTOM);
end SET_PHANTOMS;

procedure PRINT(FILE : in FILE_TYPE;
                LINE : in LINE_TYPE;
                ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK) is
  NEW_BREAK, NEW_INDEX : INTEGER;
begin
  if LINE.INDEX + ITEM'LENGTH + LINE.END_PHANTOM'LENGTH > LINE.LENGTH + 1
      then
    if LINE.INDENT + LINE.CONTINUATION_INDENT + LINE.START_PHANTOM'LENGTH +
        LINE.INDEX - LINE.BREAK + ITEM'LENGTH > LINE.LENGTH then
      raise LAYOUT_ERROR;
    end if;
    if ITEM = " " and then LINE.END_PHANTOM.all = "" then
      return;
    end if;
    PUT_LINE(FILE, LINE.DATA(1..LINE.BREAK-1) & LINE.END_PHANTOM.all);
    for I in 1..LINE.INDENT + LINE.CONTINUATION_INDENT loop
      LINE.DATA(I) := ' ';
    end loop;
    NEW_BREAK := LINE.INDENT + LINE.CONTINUATION_INDENT + 1;
    NEW_INDEX := NEW_BREAK + LINE.START_PHANTOM'LENGTH +
        LINE.INDEX - LINE.BREAK;
    LINE.DATA(NEW_BREAK..NEW_INDEX) := LINE.START_PHANTOM.all &
        LINE.DATA(LINE.BREAK..LINE.INDEX);
    LINE.BREAK := NEW_BREAK;
    LINE.INDEX := NEW_INDEX;
  end if;
  NEW_INDEX := LINE.INDEX + ITEM'LENGTH;
  LINE.DATA(LINE.INDEX..NEW_INDEX-1) := ITEM;
  LINE.INDEX := NEW_INDEX;
  if BRK = BREAK then
    LINE.BREAK := NEW_INDEX;
  end if;
  LINE.USED_YET := TRUE;
```

```
end PRINT;

procedure PRINT(FILE : in FILE_TYPE;
                ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK) is
begin
  PRINT(FILE,DEFAULT_LINE,ITEM,BRK);
end PRINT;

procedure PRINT(LINE : in LINE_TYPE;
                ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK) is
begin
  PRINT(CURRENT_OUTPUT,LINE,ITEM,BRK);
end PRINT;

procedure PRINT(ITEM : in STRING; BRK : in BREAK_TYPE := BREAK) is
begin
  PRINT(CURRENT_OUTPUT,DEFAULT_LINE,ITEM,BRK);
end PRINT;

procedure PRINT_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE) is
begin
  if LINE.INDEX /= LINE.INDENT + 1 then
    PUT_LINE(FILE,LINE.DATA(1..LINE.INDEX-1));
  end if;
  for I in 1..LINE.INDENT loop
    LINE.DATA(I) := ' ';
  end loop;
  LINE.INDEX := LINE.INDENT + 1;
  LINE.BREAK := LINE.INDEX;
end PRINT_LINE;

procedure PRINT_LINE(FILE : in FILE_TYPE) is
begin
  PRINT_LINE(FILE,DEFAULT_LINE);
end PRINT_LINE;

procedure PRINT_LINE(LINE : in LINE_TYPE) is
begin
  PRINT_LINE(CURRENT_OUTPUT,LINE);
end PRINT_LINE;

procedure PRINT_LINE is
begin
  PRINT_LINE(CURRENT_OUTPUT,DEFAULT_LINE);
end PRINT_LINE;

procedure BLANK_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE) is
begin
  if LINE.USED_YET then
    NEW_LINE(FILE);
  end if;
end BLANK_LINE;

procedure BLANK_LINE(FILE : in FILE_TYPE) is
begin
```

```
    BLANK_LINE(FILE,DEFAULT_LINE);
  end BLANK_LINE;

  procedure BLANK_LINE(LINE : in LINE_TYPE) is
  begin
    BLANK_LINE(CURRENT_OUTPUT,LINE);
  end BLANK_LINE;

  procedure BLANK_LINE is
  begin
    BLANK_LINE(CURRENT_OUTPUT,DEFAULT_LINE);
  end BLANK_LINE;

  package body INTEGER_PRINT is

    procedure PRINT(FILE : in FILE_TYPE;
                    LINE : in LINE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
      S : STRING(1..NUM'WIDTH);
      L : NATURAL;
    begin
      PRINT(S,L,ITEM);
      PRINT(FILE,LINE,S(1..L),BRK);
    end PRINT;

    procedure PRINT(FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
    begin
      PRINT(FILE,DEFAULT_LINE,ITEM,BRK);
    end PRINT;

    procedure PRINT(LINE : in LINE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
    begin
      PRINT(CURRENT_OUTPUT,LINE,ITEM,BRK);
    end PRINT;

    procedure PRINT(ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
    begin
      PRINT(CURRENT_OUTPUT,DEFAULT_LINE,ITEM,BRK);
    end PRINT;

    procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM) is
      S : constant STRING := NUM'IMAGE(ITEM);
      F : NATURAL := S'FIRST; -- Bug in DG Compiler -- S'FIRST /= 1 ! ! !
      L : NATURAL;
    begin
      if S(F) = ' ' then
        F := F + 1;
      end if;
      if TO'LENGTH < S'LAST - F + 1 then
        raise LAYOUT_EPROR;
      end if;
```

```
    L := TO'FIRST + S'LAST - F;
    TO(TO'FIRST..L) := S(F..S'LAST);
    LAST := L;
  end PRINT;

end INTEGER_PRINT;

package body FLOAT_PRINT is

  package NUM_IO is new FLOAT_IO(NUM);
    use NUM_IO;

  procedure PRINT(FILE : in FILE_TYPE;
                  LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK) is
    S : STRING(1..DEFAULT_FORE + DEFAULT_AFT + DEFAULT_EXP + 2);
    L : NATURAL;
  begin
    PRINT(S,L,ITEM);
    PRINT(FILE,LINE,S(1..L),BRK);
  end PRINT;

  procedure PRINT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK) is
  begin
    PRINT(FILE,DEFAULT_LINE,ITEM,BRK);
  end PRINT;

  procedure PRINT(LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK) is
  begin
    PRINT(CURRENT_OUTPUT,LINE,ITEM,BRK);
  end PRINT;

  procedure PRINT(ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK) is
  begin
    PRINT(CURRENT_OUTPUT,DEFAULT_LINE,ITEM,BRK);
  end PRINT;

  procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM) is
    S         : STRING(1..DEFAULT_FORE + DEFAULT_AFT + DEFAULT_EXP + 2);
    EXP       : INTEGER;
    E_INDEX   : NATURAL := S'LAST - DEFAULT_EXP;
    DOT_INDEX : NATURAL := DEFAULT_FORE + 1;
    L         : NATURAL := 0;
  begin
    PUT(S,ITEM);
    EXP := INTEGER'VALUE(S(E_INDEX+1..S'LAST));
    if EXP >= 0 and then EXP <= DEFAULT_AFT-1 then
      S(DOT_INDEX..DOT_INDEX+EXP-1) := S(DOT_INDEX+1..DOT_INDEX+EXP);
      S(DOT_INDEX+EXP) := '.';
      for I in E_INDEX..S'LAST loop
        S(I) := ' ';
```

```
        end loop;
      end if;
      for I in reverse 1..E_INDEX-1 loop
        exit when S(I) /= '0' or else S(I-1) = '.';
        S(I) := ' ';
      end loop;
      for I in S'RANGE loop
        if S(I) /= ' ' then
          L := L + 1;
          TO(L) := S(I);
        end if;
      end loop;
      LAST := L;
    exception
      when CONSTRAINT_ERROR =>
        raise LAYOUT_ERROR;
    end PRINT;

  end FLOAT_PRINT;

end TEXT_PRINT;
```

```
package DDL_DEFINITIONS is

  type TYPE_TYPE is (SUB_TYPE, REC_ORD, ENUMERATION, INT_EGER, FL_OAT,
      STR_ING);

  type TYPE_NAME_STRING is new STRING;
  type TYPE_NAME        is access TYPE_NAME_STRING;

  type TYPE_DESCRIPTOR(TY_PE : TYPE_TYPE);
  type ACCESS_TYPE_DESCRIPTOR is access TYPE_DESCRIPTOR;

  subtype ACCESS_SUBTYPE_DESCRIPTOR     is ACCESS_TYPE_DESCRIPTOR(SUB_TYPE);
  subtype ACCESS_RECORD_DESCRIPTOR      is ACCESS_TYPE_DESCRIPTOR(REC_ORD);
  subtype ACCESS_ENUMERATION_DESCRIPTOR is
                                         ACCESS_TYPE_DESCRIPTOR(ENUMERATION);
  subtype ACCESS_INTEGER_DESCRIPTOR     is ACCESS_TYPE_DESCRIPTOR(INT_EGER);
  subtype ACCESS_FLOAT_DESCRIPTOR       is ACCESS_TYPE_DESCRIPTOR(FL_OAT);
  subtype ACCESS_STRING_DESCRIPTOR      is ACCESS_TYPE_DESCRIPTOR(STR_ING);

  type COMPONENT_NAME_STRING is new STRING;
  type COMPONENT_NAME        is access COMPONENT_NAME_STRING;

  type COMPONENT_DESCRIPTOR;
  type ACCESS_COMPONENT_DESCRIPTOR is access COMPONENT_DESCRIPTOR;

  type COMPONENT_DESCRIPTOR is
    record
      NEXT_COMPONENT,
      PREVIOUS_COMPONENT : ACCESS_COMPONENT_DESCRIPTOR;
      NAME               : COMPONENT_NAME;
      TY_PE,
      PARENT_RECORD      : ACCESS_TYPE_DESCRIPTOR;
    end record;

  type SUBRECORD_INDICATOR is new BOOLEAN;

  type LITERAL_DESCRIPTOR;
  type ACCESS_LITERAL_DESCRIPTOR is access LITERAL_DESCRIPTOR;

  type ENUMERATION_NAME_STRING is new STRING;
  type ENUMERATION_NAME        is access ENUMERATION_NAME_STRING;

  type ENUMERATION_POS is new NATURAL;

  type LITERAL_DESCRIPTOR is
    record
      NEXT_LITERAL,
      PREVIOUS_LITERAL : ACCESS_LITERAL_DESCRIPTOR;
      NAME             : ENUMERATION_NAME;
      POS              : ENUMERATION_POS;
      PARENT_TYPE      : ACCESS_TYPE_DESCRIPTOR;
    end record;

  type STRING_LENGTH is new NATURAL;

  type TYPE_DESCRIPTOR(TY_PE : TYPE_TYPE) is
    record
```

```
        NAME         : TYPE_NAME;
        NEXT_TYPE,
        PREVIOUS_TYPE,
        FIRST_SUBTYPE,
        LAST_SUBTYPE : ACCESS_TYPE_DESCRIPTOR;
        case TY_PE is
          when SUB_TYPE =>
            PARENT_TYPE,
            TOP_TYPE,
            NEXT_SUBTYPE,
            PREVIOUS_SUBTYPE : ACCESS_TYPE_DESCRIPTOR;
          when REC_ORD =>
            FIRST_COMPONENT,
            LAST_COMPONENT    : ACCESS_COMPONENT_DESCRIPTOR;
            IS_SUBRECORD      : SUBRECORD_INDICATOR := FALSE;
          when ENUMERATION =>
            FIRST_LITERAL,
            LAST_LITERAL      : ACCESS_LITERAL_DESCRIPTOR;
            LAST_POS          : ENUMERATION_POS := 0;
            MAX_LENGTH        : NATURAL := 0;
          when INT_EGER | FL_OAT =>
            null;
          when STR_ING =>
            LENGTH : STRING_LENGTH;
        end case;
      end record;

  end DDL_DEFINITIONS;
```

```
with DDL_DEFINITIONS;
  use DDL_DEFINITIONS;

package LIST_UTILITIES is

  function FIRST_TYPE_DESCRIPTOR return ACCESS_TYPE_DESCRIPTOR;

  function FIND_TYPE_DESCRIPTOR(NAME : TYPE_NAME_STRING)
      return ACCESS_TYPE_DESCRIPTOR;

  procedure ADD_TYPE(T : ACCESS_TYPE_DESCRIPTOR);

  procedure ADD_SUBTYPE(PARENT : ACCESS_TYPE_DESCRIPTOR;
                        CHILD  : ACCESS_SUBTYPE_DESCRIPTOR);

  procedure ADD_LITERAL(PARENT : ACCESS_ENUMERATION_DESCRIPTOR;
                        CHILD  : ACCESS_LITERAL_DESCRIPTOR);

  procedure ADD_COMPONENT(PARENT : ACCESS_RECORD_DESCRIPTOR;
                          CHILD  : ACCESS_COMPONENT_DESCRIPTOR);

end LIST_UTILITIES;

package body LIST_UTILITIES is

  TYPE_DESCRIPTOR_0,                             -- type listhead -- first & last
  TYPE_DESCRIPTOR_9 : ACCESS_TYPE_DESCRIPTOR;  --  type descriptors

  function FIRST_TYPE_DESCRIPTOR return ACCESS_TYPE_DESCRIPTOR is
  begin
    return TYPE_DESCRIPTOR_0;
  end FIRST_TYPE_DESCRIPTOR;

  function FIND_TYPE_DESCRIPTOR(NAME : TYPE_NAME_STRING)
      return ACCESS_TYPE_DESCRIPTOR is
    T : ACCESS_TYPE_DESCRIPTOR := TYPE_DESCRIPTOR_0;
  begin
    while T.NAME.all /= NAME loop
      T := T.NEXT_TYPE; -- CONSTRAINT_ERROR if non-existent type name
    end loop;
    return T;
  end FIND_TYPE_DESCRIPTOR;

  procedure ADD_TYPE(T : ACCESS_TYPE_DESCRIPTOR) is
  begin
    if TYPE_DESCRIPTOR_9 = null then
      TYPE_DESCRIPTOR_0 := T;
    else
      TYPE_DESCRIPTOR_9.NEXT_TYPE := T;
    end if;
    T.PREVIOUS_TYPE := TYPE_DESCRIPTOR_9;
    TYPE_DESCRIPTOR_9 := T;
    T.NEXT_TYPE := null;
  end ADD_TYPE;

  procedure ADD_SUBTYPE(PARENT : ACCESS_TYPE_DESCRIPTOR;
                        CHILD  : ACCESS_SUBTYPE_DESCRIPTOR) is
```

```
    begin
      if PARENT.LAST_SUBTYPE = null then
        PARENT.FIRST_SUBTYPE := CHILD;
      else
        PARENT.LAST_SUBTYPE.NEXT_SUBTYPE := CHILD;
      end if;
      CHILD.PREVIOUS_SUBTYPE := PARENT.LAST_SUBTYPE;
      PARENT.LAST_SUBTYPE := CHILD;
      CHILD.NEXT_SUBTYPE := null;
      CHILD.PARENT_TYPE := PARENT;
    end ADD_SUBTYPE;

    procedure ADD_LITERAL(PARENT : ACCESS_ENUMERATION_DESCRIPTOR;
                          CHILD  : ACCESS_LITERAL_DESCRIPTOR) is
    begin
      if PARENT.LAST_LITERAL = null then
        PARENT.FIRST_LITERAL := CHILD;
      else
        PARENT.LAST_LITERAL.NEXT_LITERAL := CHILD;
      end if;
      CHILD.PREVIOUS_LITERAL := PARENT.LAST_LITERAL;
      PARENT.LAST_LITERAL := CHILD;
      CHILD.NEXT_LITERAL := null;
      CHILD.PARENT_TYPE := PARENT;
    end ADD_LITERAL;

    procedure ADD_COMPONENT(PARENT : ACCESS_RECORD_DESCRIPTOR;
                            CHILD  : ACCESS_COMPONENT_DESCRIPTOR) is
    begin
      if PARENT.LAST_COMPONENT = null then
        PARENT.FIRST_COMPONENT := CHILD;
      else
        PARENT.LAST_COMPONENT.NEXT_COMPONENT := CHILD;
      end if;
      CHILD.PREVIOUS_COMPONENT := PARENT.LAST_COMPONENT;
      PARENT.LAST_COMPONENT := CHILD;
      CHILD.NEXT_COMPONENT := null;
      CHILD.PARENT_RECORD := PARENT;
    end ADD_COMPONENT;

  end LIST_UTILITIES;
```

```
with DDL_DEFINITIONS, LIST_UTILITIES, TOKEN_INPUT;
  use DDL_DEFINITIONS, LIST_UTILITIES, TOKEN_INPUT;

package READ_DDL is

  procedure SCAN_DDL(PACKAGE_NAME : out STRING;
                     LAST         : out POSITIVE);

end READ_DDL;

package body READ_DDL is

  procedure PROCESS_DERIVED_TYPE(NEW_NAME : TYPE_NAME) is
    KEYWORD  : STRING(1..7);
    LAST     : POSITIVE;
    STR_LAST : STRING_LENGTH;
  begin
    GET_STRING(KEYWORD,LAST);
    if KEYWORD(1..LAST) = "INTEGER" then
      ADD_TYPE ( new TYPE_DESCRIPTOR'(TY_PE => INT_EGER, NAME => NEW_NAME,
          others => null) );
      GOBBLE(";");
    elsif KEYWORD(1..LAST) = "FLOAT" then
      ADD_TYPE ( new TYPE_DESCRIPTOR'(TY_PE => FL_OAT,   NAME => NEW_NAME,
          others => null) );
      GOBBLE(";");
    elsif KEYWORD(1..LAST) = "STRING" then
      GOBBLE("("); GOBBLE("1"); GOBBLE("..");
      STR_LAST := STRING_LENGTH(GET_INTEGER);
      ADD_TYPE ( new TYPE_DESCRIPTOR'(TY_PE => STR_ING,  NAME => NEW_NAME,
          LENGTH => STR_LAST, others => null) );
      GOBBLE(");");
    else
      raise CONSTRAINT_ERROR; -- unrecognized type keyword
    end if;
  end PROCESS_DERIVED_TYPE;

  procedure PROCESS_ENUMERATION_TYPE(NEW_NAME : TYPE_NAME) is
    PARENT    : ACCESS_ENUMERATION_DESCRIPTOR;
    LITERAL   : ENUMERATION_NAME_STRING(1..80);
    LAST      : POSITIVE;
    DELIMITER : STRING(1..2);
  begin
    PARENT := new TYPE_DESCRIPTOR'(TY_PE => ENUMERATION, NAME => NEW_NAME,
        LAST_POS => 0, MAX_LENGTH => 0, FIRST_LITERAL | LAST_LITERAL => null,
        others => null);
    ADD_TYPE(PARENT);
    loop
      GET_STRING(STRING(LITERAL),LAST);
      PARENT.LAST_POS := PARENT.LAST_POS + 1;
      if LAST > PARENT.MAX_LENGTH then
        PARENT.MAX_LENGTH := LAST;
      end if;
      ADD_LITERAL ( PARENT, new LITERAL_DESCRIPTOR'(
          NAME => new ENUMERATION_NAME_STRING'(LITERAL(1..LAST)),
          POS  => PARENT.LAST_POS, PARENT_TYPE => PARENT, others => null) );
      GET_STRING(DELIMITER,LAST);
```

```
      if DELIMITER(1..LAST) = ");" then
        exit;
      elsif DELIMITER(1..LAST) /= "," then
        raise CONSTRAINT_ERROR; -- invalid enumeration literal list
      end if;
    end loop;
  end PROCESS_ENUMERATION_TYPE;

  procedure PROCESS_RECORD_TYPE(NEW_NAME : TYPE_NAME) is
    FIELD_TYPE_NAME : TYPE_NAME_STRING(1..80);
    FIELD_NAME      : COMPONENT_NAME_STRING(1..80);
    FIELD_TYPE      : ACCESS_TYPE_DESCRIPTOR;
    PARENT          : ACCESS_RECORD_DESCRIPTOR;
    FIELD_TYPE_LAST,
    FIELD_LAST      : POSITIVE;
  begin
    PARENT := new TYPE_DESCRIPTOR'(TY_PE => REC_ORD, NAME => NEW_NAME,
        IS_SUBRECORD => FALSE, FIRST_COMPONENT | LAST_COMPONENT => null,
        others => null);
    ADD_TYPE(PARENT);
    loop
      GET_STRING(STRING(FIELD_NAME),FIELD_LAST);
      if FIELD_NAME(1..FIELD_LAST) = "end" then
        GOBBLE("record"); GOBBLE(";");
        exit;
      end if;
      GOBBLE(":");
      GET_STRING(STRING(FIELD_TYPE_NAME),FIELD_TYPE_LAST);
      FIELD_TYPE := FIND_TYPE_DESCRIPTOR(FIELD_TYPE_NAME(1..FIELD_TYPE_LAST));
      if FIELD_TYPE.TY_PE = REC_ORD then
        FIELD_TYPE.IS_SUBRECORD := TRUE;
      end if;
      ADD_COMPONENT ( PARENT, new COMPONENT_DESCRIPTOR' (
          NAME  => new COMPONENT_NAME_STRING'(FIELD_NAME(1..FIELD_LAST)),
          TY_PE => FIELD_TYPE, PARENT_RECORD => PARENT, others => null ) );
      GOBBLE(";");
    end loop;
  end PROCESS_RECORD_TYPE;

  procedure PROCESS_TYPE is
    NAME_STRING    : TYPE_NAME_STRING(1..80);
    NAME           : TYPE_NAME;
    LAST           : POSITIVE;
    TYPE_INDICATOR : STRING(1..6);
  begin
    GET_STRING(STRING(NAME_STRING),LAST);
    NAME := new TYPE_NAME_STRING'(NAME_STRING(1..LAST));
    GOBBLE("is");
    GET_STRING(TYPE_INDICATOR,LAST);
    if TYPE_INDICATOR(1..LAST) = "(" then
      PROCESS_ENUMERATION_TYPE(NAME);
    elsif TYPE_INDICATOR(1..LAST) = "new" then
      PROCESS_DERIVED_TYPE(NAME);
    elsif TYPE_INDICATOR(1..LAST) = "record" then
      PROCESS_RECORD_TYPE(NAME);
    else
      raise CONSTRAINT_ERROR; -- unrecognized type keyword/indicator
```

```
    end if;
  end PROCESS_TYPE;

  procedure PROCESS_SUBTYPE is
    CHILD_NAME,
    PARENT_NAME         : TYPE_NAME_STRING(1..80);
    CHILD_LAST,
    PARENT_LAST         : POSITIVE;
    PARENT_DESCRIPTOR : ACCESS_TYPE_DESCRIPTOR;
    CHILD_DESCRIPTOR  : ACCESS_SUBTYPE_DESCRIPTOR;
  begin
    GET_STRING(STRING(CHILD_NAME),CHILD_LAST);
    GOBBLE("is");
    GET_STRING(STRING(PARENT_NAME),PARENT_LAST);
    GOBBLE(";");
    PARENT_DESCRIPTOR := FIND_TYPE_DESCRIPTOR(PARENT_NAME(1..PARENT_LAST));
    CHILD_DESCRIPTOR := new TYPE_DESCRIPTOR' (
        TY_PE  => SUB_TYPE,
        NAME   => new TYPE_NAME_STRING'(CHILD_NAME(1..CHILD_LAST)),
        others => null );
    ADD_TYPE(CHILD_DESCRIPTOR);
    ADD_SUBTYPE(PARENT_DESCRIPTOR, CHILD_DESCRIPTOR);
    if PARENT_DESCRIPTOR.TY_PE = SUB_TYPE then
      CHILD_DESCRIPTOR.TOP_TYPE := PARENT_DESCRIPTOR.TOP_TYPE;
    else
      CHILD_DESCRIPTOR.TOP_TYPE := PARENT_DESCRIPTOR;
    end if;
  end PROCESS_SUBTYPE;

  procedure SCAN_DDL(PACKAGE_NAME : out STRING;
                     LAST         : out POSITIVE) is
    KEYWORD : STRING(1..7);
    KLAST   : POSITIVE;
  begin
    GOBBLE("package");
    GET_STRING(PACKAGE_NAME,LAST);
    GOBBLE("is");
    loop
      GET_STRING(KEYWORD,KLAST);
      if KEYWORD(1..KLAST) = "type" then
        PROCESS_TYPE;
      elsif KEYWORD(1..KLAST) = "subtype" then
        PROCESS_SUBTYPE;
      elsif KEYWORD(1..KLAST) = "end" then
        exit;
      else
        raise CONSTRAINT_ERROR; -- unrecognized keyword
      end if;
    end loop;
  end SCAN_DDL;

end READ_DDL;
```

```
with DDL_DEFINITIONS, TEXT_PRINT;
  use DDL_DEFINITIONS, TEXT_PRINT;

package SHOW_DDL is

  package INT_PRINT is new INTEGER_PRINT(INTEGER);
    use INT_PRINT;

  procedure DISPLAY_DDL(PACKAGE_NAME : STRING);

  procedure PRINT_ENUMERATION_LITERALS(L : ACCESS_LITERAL_DESCRIPTOR);

  function MAX_COMPONENT_NAME_LENGTH(C : ACCESS_COMPONENT_DESCRIPTOR) return
      NATURAL;

end SHOW_DDL;

with DDL_DEFINITIONS, LIST_UTILITIES, TEXT_PRINT;
  use DDL_DEFINITIONS, LIST_UTILITIES, TEXT_PRINT;

package body SHOW_DDL is

  procedure SHOW_SUBTYPE_TREE(T : ACCESS_TYPE_DESCRIPTOR; LEVEL : NATURAL) is
  begin
    if T /= null then
      PRINT("-- ",NO_BREAK);
      for I in 1..LEVEL loop
        PRINT("   ",NO_BREAK);
      end loop;
      PRINT(STRING(T.NAME.all)); PRINT_LINE;
      SHOW_SUBTYPE_TREE(T.FIRST_SUBTYPE,LEVEL+1);
      if T.TY_PE = SUB_TYPE then
        SHOW_SUBTYPE_TREE(T.NEXT_SUBTYPE,LEVEL);
      end if;
    end if;
  end SHOW_SUBTYPE_TREE;

  procedure SHOW_SUBTYPES is
    CURRENT_TYPE : ACCESS_TYPE_DESCRIPTOR := FIRST_TYPE_DESCRIPTOR;
  begin
    PRINT("-- subtype tree"); PRINT_LINE; BLANK_LINE;
    while CURRENT_TYPE /= null loop
      if CURRENT_TYPE.TY_PE /= SUB_TYPE then
        SHOW_SUBTYPE_TREE(CURRENT_TYPE,0);
        BLANK_LINE;
      end if;
      CURRENT_TYPE := CURRENT_TYPE.NEXT_TYPE;
    end loop;
  end SHOW_SUBTYPES;

  function BLANK_LINE_FOLLOWS(T : ACCESS_TYPE_DESCRIPTOR) return BOOLEAN is
  begin
    if T.TY_PE = REC_ORD or else T.NEXT_TYPE = null then
      return TRUE;
    elsif T.NEXT_TYPE.TY_PE /= SUB_TYPE then
      case T.TY_PE is
        when INT_EGER | FL_OAT | STR_ING =>
```

```
          if T.TY_PE /= T.NEXT_TYPE.TY_PE or else
              ( T.NEXT_TYPE.NEXT_TYPE /= null and then
                  T.NEXT_TYPE.NEXT_TYPE.TY_PE = SUB_TYPE ) then
            return TRUE;
          end if;
        when others =>
          return TRUE;
      end case;
    end if;
    return FALSE;
  end BLANK_LINE_FOLLOWS;

  function MAX_TYPE_NAME_LENGTH(T : ACCESS_TYPE_DESCRIPTOR) return NATURAL is
    LENGTH : NATURAL := 0;
    Q      : ACCESS_TYPE_DESCRIPTOR := T;
  begin
    while Q /= null loop
      if Q.NAME'LAST > LENGTH then
        LENGTH := Q.NAME'LAST;
      end if;
      exit when BLANK_LINE_FOLLOWS(Q);
      Q := Q.NEXT_TYPE;
    end loop;
    return LENGTH;
  end MAX_TYPE_NAME_LENGTH;

  function MAX_COMPONENT_NAME_LENGTH(C : ACCESS_COMPONENT_DESCRIPTOR) return
      NATURAL is
    LENGTH : NATURAL := 0;
    D      : ACCESS_COMPONENT_DESCRIPTOR := C;
  begin
    while D /= null loop
      if D.NAME'LAST > LENGTH then
        LENGTH := D.NAME'LAST;
      end if;
      D := D.NEXT_COMPONENT;
    end loop;
    return LENGTH;
  end MAX_COMPONENT_NAME_LENGTH;

  procedure PRINT_TYPE_IS(T      : ACCESS_TYPE_DESCRIPTOR;
                          LENGTH : NATURAL) is
  begin
    if T.TY_PE = SUB_TYPE then
      PRINT("subtype ",NO_BREAK);
    else
      PRINT("type ",NO_BREAK);
    end if;
    PRINT(STRING(T.NAME.all),NO_BREAK);
    if T.TY_PE = SUB_TYPE or else T.NEXT_TYPE = null or else
        T.NEXT_TYPE.TY_PE /= SUB_TYPE then
      for I in 1..LENGTH-T.NAME'LAST loop
        PRINT(" ",NO_BREAK);
      end loop;
    end if;
    PRINT(" is",NO_BREAK);
  end PRINT_TYPE_IS;
```

```
procedure PRINT_ENUMERATION_LITERALS(L : ACCESS_LITERAL_DESCRIPTOR) is
          M : ACCESS_LITERAL_DESCRIPTOR := L;
begin
  loop
    PRINT(STRING(M.NAME.all),NO_BREAK);
    M := M.NEXT_LITERAL;
    if M = null then
      exit;
    else
      PRINT(",");
    end if;
  end loop;
end PRINT_ENUMERATION_LITERALS;

procedure PRINT_RECORD_COMPONENTS(C : ACCESS_COMPONENT_DESCRIPTOR) is
  D : ACCESS_COMPONENT_DESCRIPTOR := C;
  LENGTH : NATURAL := MAX_COMPONENT_NAME_LENGTH(D);
begin
  while D /= null loop
    PRINT(STRING(D.NAME.all),NO_BREAK);
    for I in 1..LENGTH-D.NAME'LAST loop
      PRINT(" ",NO_BREAK);
    end loop;
    PRINT(" : " & STRING(D.TY_PE.NAME.all) & ";"); PRINT_LINE;
    D := D.NEXT_COMPONENT;
  end loop;
end PRINT_RECORD_COMPONENTS;

procedure SHOW_SOURCE is
  CURRENT_TYPE : ACCESS_TYPE_DESCRIPTOR := FIRST_TYPE_DESCRIPTOR;
  NAME_LENGTH  : NATURAL := MAX_TYPE_NAME_LENGTH(CURRENT_TYPE);
begin
  while CURRENT_TYPE /= null loop
    PRINT_TYPE_IS(CURRENT_TYPE,NAME_LENGTH);
    case CURRENT_TYPE.TY_PE is
      when SUB_TYPE =>
        PRINT(" ",NO_BREAK);
        PRINT(STRING(CURRENT_TYPE.PARENT_TYPE.NAME.all),NO_BREAK);
      when REC_ORD =>
        PRINT_LINE; PRINT("  record"); PRINT_LINE; SET_INDENT(6);
        PRINT_RECORD_COMPONENTS(CURRENT_TYPE.FIRST_COMPONENT);
        SET_INDENT(2); PRINT("  end record");
      when ENUMERATION =>
        PRINT(" (",NO_BREAK);
        PRINT_ENUMERATION_LITERALS(CURRENT_TYPE.FIRST_LITERAL);
        PRINT(")",NO_BREAK);
      when INT_EGER =>
        PRINT(" new INTEGER",NO_BREAK);
      when FL_OAT =>
        PRINT(" new FLOAT",NO_BREAK);
      when STR_ING =>
        PRINT(" new STRING(1..",NO_BREAK);
        PRINT(INTEGER(CURRENT_TYPE.LENGTH),NO_BREAK);
        PRINT(")",NO_BREAK);
    end case;
    PRINT(";"); PRINT_LINE;
    if BLANK_LINE_FOLLOWS(CURRENT_TYPE) then
```

```
        BLANK_LINE;
        NAME_LENGTH := MAX_TYPE_NAME_LENGTH(CURRENT_TYPE.NEXT_TYPE);
      end if;
      CURRENT_TYPE := CURRENT_TYPE.NEXT_TYPE;
    end loop;
  end SHOW_SOURCE;

  procedure DISPLAY_DDL(PACKAGE_NAME : STRING) is
  begin
    SET_INDENT(0); SET_CONTINUATION_INDENT(2);
    PRINT("package " & PACKAGE_NAME & " is"); PRINT_LINE; BLANK_LINE;
    SET_INDENT(2);
    SHOW_SUBTYPES;
    SHOW_SOURCE;
    SET_INDENT(0);
    PRINT("end " & PACKAGE_NAME & ";"); PRINT_LINE;
  end DISPLAY_DDL;

end SHOW_DDL;
```

```
package SIMPLE_DDL is

  procedure GENERATE_SIMPLE_DDL;

end SIMPLE_DDL;

with DDL_DEFINITIONS, LIST_UTILITIES, SHOW_DDL, TEXT_PRINT;
  use DDL_DEFINITIONS, LIST_UTILITIES, SHOW_DDL, TEXT_PRINT;

package body SIMPLE_DDL is

  use SHOW_DDL.INT_PRINT;

  procedure PRINT_FIELDS(C : ACCESS_COMPONENT_DESCRIPTOR) is
    D : ACCESS_COMPONENT_DESCRIPTOR := C;
    T : ACCESS_TYPE_DESCRIPTOR;
  begin
    while D /= null loop
      T := D.TY_PE;
      if T.TY_PE = SUB_TYPE then
        T := T.TOP_TYPE;
      end if;
      case T.TY_PE is
        when SUB_TYPE =>
          raise PROGRAM_ERROR; -- internal error due to if above
        when REC_ORD =>
          PRINT_FIELDS(T.FIRST_COMPONENT);
        when ENUMERATION =>
          PRINT("FIELD " & STRING(D.NAME.all) & " STRING ",NO_BREAK);
          PRINT(T.MAX_LENGTH + ENUMERATION_POS'IMAGE(T.LAST_POS)'LENGTH - 1);
          PRINT_LINE;
        when INT_EGER =>
          PRINT("FIELD " & STRING(D.NAME.all) & " INTEGER 6"); PRINT_LINE;
        when FL_OAT =>
          PRINT("FIELD " & STRING(D.NAME.all) & " FLOAT 7"); PRINT_LINE;
        when STR_ING =>
          PRINT("FIELD " & STRING(D.NAME.all) & " STRING ",NO_BREAK);
          PRINT(INTEGER(T.LENGTH)); PRINT_LINE;
      end case;
      D := D.NEXT_COMPONENT;
    end loop;
  end PRINT_FIELDS;

  procedure GENERATE_SIMPLE_DDL is
    CURRENT_TYPE : ACCESS_TYPE_DESCRIPTOR := FIRST_TYPE_DESCRIPTOR;
  begin
    SET_INDENT(0); SET_CONTINUATION_INDENT(2);
    while CURRENT_TYPE /= null loop
      if CURRENT_TYPE.TY_PE = REC_ORD and then
          CURRENT_TYPE.IS_SUBRECORD = FALSE then
        PRINT("TABLE " & STRING(CURRENT_TYPE.NAME.all)); PRINT_LINE;
        BLANK_LINE;
        PRINT_FIELDS(CURRENT_TYPE.FIRST_COMPONENT);
        BLANK_LINE;
      end if;
      CURRENT_TYPE := CURRENT_TYPE.NEXT_TYPE;
    end loop;
```

```
    PRINT("END"); PRINT_LINE;
  end GENERATE_SIMPLE_DDL;

end SIMPLE_DDL;
```

```
package DAMES_DDL is

  procedure GENERATE_DAMES_DDL;

end DAMES_DDL;

with DDL_DEFINITIONS, LIST_UTILITIES, SHOW_DDL, TEXT_PRINT;
  use DDL_DEFINITIONS, LIST_UTILITIES, SHOW_DDL, TEXT_PRINT;

package body DAMES_DDL is

  use SHOW_DDL.INT_PRINT;

  START_PHANTOM : constant PHANTOM_TYPE := MAKE_PHANTOM("""");
  END_PHANTOM   : constant PHANTOM_TYPE := MAKE_PHANTOM(""" &");

  procedure PRINT_FIELD_NAME(C          : in ACCESS_COMPONENT_DESCRIPTOR;
                             FIRST_TIME : in out BOOLEAN;
                             SEPARATOR  : in STRING) is
  begin
    if FIRST_TIME then
      FIRST_TIME := FALSE;
    else
      PRINT(SEPARATOR & """ &"); PRINT_LINE;
    end if;
    PRINT("""" & STRING(C.NAME.all),NO_BREAK);
  end PRINT_FIELD_NAME;

  procedure PRINT_FIELDS(C          : ACCESS_COMPONENT_DESCRIPTOR;
                         FIRST_TIME : BOOLEAN := TRUE;
                         SEPARATOR  : STRING  := ";") is
    D  : ACCESS_COMPONENT_DESCRIPTOR := C;
    T  : ACCESS_TYPE_DESCRIPTOR;
    FT : BOOLEAN := FIRST_TIME;
  begin
    while D /= null loop
      T := D.TY_PE;
      if T.TY_PE = SUB_TYPE then
        T := T.TOP_TYPE;
      end if;
      case T.TY_PE is
        when SUB_TYPE =>
          raise PROGRAM_ERROR; -- internal error due to if above
        when REC_ORD =>
          if D.PARENT_RECORD.IS_SUBRECORD = TRUE then
            PRINT_FIELDS(T.FIRST_COMPONENT,FT,SEPARATOR); FT := FALSE;
          else
            PRINT_FIELD_NAME(D,FT,SEPARATOR); PRINT(" "" &"); PRINT_LINE;
            SET_INDENT(5);
            PRINT_FIELDS(T.FIRST_COMPONENT,TRUE,",");
            SET_INDENT(3);
          end if;
        when ENUMERATION =>
          PRINT_FIELD_NAME(D,FT,SEPARATOR); PRINT(' (",NO_BREAK);
          PRINT_ENUMERATION_LITERALS(T.FIRST_LITERAL); PRINT(")",NO_BREAK);
        when INT_EGER =>
          PRINT_FIELD_NAME(D,FT,SEPARATOR); PRINT(" INTEGER");
```

```
          when FL_OAT =>
            PRINT_FIELD_NAME(D,FT,SEPARATOR); PRINT(" FLOAT");
          when STR_ING =>
            PRINT_FIELD_NAME(D,FT,SEPARATOR); PRINT(" STRING 1..",NO_BREAK);
            PRINT(INTEGER(T.LENGTH));
        end case;
        D := D.NEXT_COMPONENT;
      end loop;
    end PRINT_FIELDS;

    procedure GENERATE_DAMES_DDL is
      CURRENT_TYPE : ACCESS_TYPE_DESCRIPTOR := FIRST_TYPE_DESCRIPTOR;
      FIRST_TIME   : BOOLEAN := TRUE;
    begin
      SET_CONTINUATION_INDENT(2); SET_PHANTOMS(START_PHANTOM,END_PHANTOM);
      while CURRENT_TYPE /= null loop
        if CURRENT_TYPE.TY_PE = REC_ORD and then
            CURRENT_TYPE.IS_SUBRECORD = FALSE then
          if FIRST_TIME then
            FIRST_TIME := FALSE;
          else
            BLANK_LINE;
          end if;
          SET_INDENT(2);
          PRINT("DEFINE_TABLE(""" & STRING(CURRENT_TYPE.NAME.all) & """,");
          PRINT_LINE; SET_INDENT(3);
          PRINT_FIELDS(CURRENT_TYPE.FIRST_COMPONENT);
          PRINT(""");"); PRINT_LINE;
        end if;
        CURRENT_TYPE := CURRENT_TYPE.NEXT_TYPE;
      end loop;
      SET_INDENT(0); SET_PHANTOMS(NULL_PHANTOM,NULL_PHANTOM);
    end GENERATE_DAMES_DDL;

  end DAMES_DDL;
```

```
with DAMES_DDL, READ_DDL, SHOW_DDL, SIMPLE_DDL, TEXT_IO, TEXT_PRINT,
    TOKEN_INPUT;
  use DAMES_DDL, READ_DDL, SHOW_DDL, SIMPLE_DDL, TEXT_IO, TEXT_PRINT,
      TOKEN_INPUT;

procedure MAIN is

  LINE         : LINE_TYPE;
  PACKAGE_NAME : STRING(1..80);
  LAST         : NATURAL;

  procedure PRINT_RULE is
  begin
    PRINT("---------------------------------------" &
          "---------------------------------------"); PRINT_LINE;
  end PRINT_RULE;

begin
  SET_STREAM(CREATE_STREAM(80)); OPEN_INPUT("BOATS.ADA");
  CREATE_LINE(LINE,79); SET_LINE(LINE);
  SCAN_DDL(PACKAGE_NAME,LAST);
  DISPLAY_DDL(PACKAGE_NAME(1..LAST)); PRINT_RULE;
  GENERATE_SIMPLE_DDL; PRINT_RULE;
  GENERATE_DAMES_DDL;
  CLOSE_INPUT;
end MAIN;
```

```
package BOATS is

  type OCEAN_NAME is (INDIAN,ATLANTIC,PACIFIC,MEDITERRANEAN,ARCTIC,CARRIBEAN,
      SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);
  subtype UNIQUE_OCEAN_NAME is OCEAN_NAME;

  type ANALYST_NAME is new STRING(1..20);
  subtype UNIQUE_ANALYST_NAME is ANALYST_NAME;
  subtype MANAGER_NAME        is UNIQUE_ANALYST_NAME;

  type ANALYST_SALARY is new FLOAT;

  type SHIP_NAME is new STRING(1..15);
  subtype UNIQUE_SHIP_NAME is SHIP_NAME;

  type POSITION_LAT  is new STRING(1..7);
  type POSITION_LONG is new STRING(1..8);

  type POSITION_LATLONG is
    record
      LAT  : POSITION_LAT;
      LONG : POSITION_LONG;
    end record;

  type SHIP_TYPE is (CARRIER,DESTROYER);

  type CREW_SPECIALTY is (COOK,SHUFFLEBOARD_TEACHER);

  type SHIP_CREW is
    record
      TY_PE     : SHIP_TYPE;
      SPECIALTY : CREW_SPECIALTY;
    end record;

  subtype UNIQUE_SHIP_CREW is SHIP_CREW;

  type CREW_COUNT is new INTEGER;

  type OCEAN is
    record
      NAME    : UNIQUE_OCEAN_NAME;
      ANALYST : UNIQUE_ANALYST_NAME;
    end record;

  type ANALYST is
    record
      NAME    : UNIQUE_ANALYST_NAME;
      SALARY  : ANALYST_SALARY;
      MANAGER : MANAGER_NAME;
    end record;

  type SHIP is
    record
      NAME    : UNIQUE_SHIP_NAME;
      OCEAN   : OCEAN_NAME;
      LATLONG : POSITION_LATLONG;
      TY_PE   : SHIP_TYPE;
```

```
        end record;

      type CREW is
        record
          KEY    : UNIQUE_SHIP_CREW;
          NUMBER : CREW_COUNT;
        end record;

      subtype SAMPLE_FIRST_LEVEL_SUBTYPE   is ANALYST_NAME;
      subtype SAMPLE_SECOND_LEVEL_SUBTYPE  is SAMPLE_FIRST_LEVEL_SUBTYPE;
      subtype ANOTHER_SECOND_LEVEL_SUBTYPE is SAMPLE_FIRST_LEVEL_SUBTYPE;

      type SAMPLE_SECOND_LEVEL_RECORD is
        record
          FIRST_LEVEL_RECORD : POSITION_LATLONG;
          SCALAR_2           : OCEAN_NAME;
        end record;

      type SAMPLE_THIRD_LEVEL_RECORD is
        record
          SECOND_LEVEL_RECORD : SAMPLE_SECOND_LEVEL_RECORD;
          FIRST_LEVEL_RECORD  : SHIP_CREW;
          SCALAR_3            : CREW_COUNT;
        end record;

    end BOATS;
```

```
-) xeq main
package BOATS is

  -- subtype tree

  -- OCEAN_NAME
  --   UNIQUE_OCEAN_NAME

  -- ANALYST_NAME
  --   UNIQUE_ANALYST_NAME
  --     MANAGER_NAME
  --   SAMPLE_FIRST_LEVEL_SUBTYPE
  --     SAMPLE_SECOND_LEVEL_SUBTYPE
  --     ANOTHER_SECOND_LEVEL_SUBTYPE

  -- ANALYST_SALARY

  -- SHIP_NAME
  --   UNIQUE_SHIP_NAME

  -- POSITION_LAT

  -- POSITION_LONG

  -- POSITION_LATLONG

  -- SHIP_TYPE

  -- CREW_SPECIALTY

  -- SHIP_CREW
  --   UNIQUE_SHIP_CREW

  -- CREW_COUNT

  -- OCEAN

  -- ANALYST

  -- SHIP

  -- CREW

  -- SAMPLE_SECOND_LEVEL_RECORD

  -- SAMPLE_THIRD_LEVEL_RECORD

  type OCEAN_NAME is (INDIAN ATLANTIC,PACIFIC,MEDITERRANEAN,ARCTIC,CARRIBEAN,
    SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);
  subtype UNIQUE_OCEAN_NAME is OCEAN_NAME;

  type ANALYST_NAME is new STRING(1..20);
  subtype UNIQUE_ANALYST_NAME is ANALYST_NAME;
  subtype MANAGER_NAME        is UNIQUE_ANALYST_NAME;

  type ANALYST_SALARY is new FLOAT;
```

```
type SHIP_NAME is new STRING(1..15);
subtype UNIQUE_SHIP_NAME is SHIP_NAME;

type POSITION_LAT  is new STRING(1..7);
type POSITION_LONG is new STRING(1..8);

type POSITION_LATLONG is
  record
    LAT  : POSITION_LAT;
    LONG : POSITION_LONG;
  end record;

type SHIP_TYPE is (CARRIER,DESTROYER);

type CREW_SPECIALTY is (COOK,SHUFFLEBOARD_TEACHER);

type SHIP_CREW is
  record
    TY_PE     : SHIP_TYPE;
    SPECIALTY : CREW_SPECIALTY;
  end record;

subtype UNIQUE_SHIP_CREW is SHIP_CREW;

type CREW_COUNT is new INTEGER;

type OCEAN is
  record
    NAME    : UNIQUE_OCEAN_NAME;
    ANALYST : UNIQUE_ANALYST_NAME;
  end record;

type ANALYST is
  record
    NAME    : UNIQUE_ANALYST_NAME;
    SALARY  : ANALYST_SALARY;
    MANAGER : MANAGER_NAME;
  end record;

type SHIP is
  record
    NAME    : UNIQUE_SHIP_NAME;
    OCEAN   : OCEAN_NAME;
    LATLONG : POSITION_LATLONG;
    TY_PE   : SHIP_TYPE;
  end record;

type CREW is
  record
    KEY    : UNIQUE_SHIP_CREW;
    NUMBER : CREW_COUNT;
  end record;

subtype SAMPLE_FIRST_LEVEL_SUBTYPE   is ANALYST_NAME;
subtype SAMPLE_SECOND_LEVEL_SUBTYPE  is SAMPLE_FIRST_LEVEL_SUBTYPE;
subtype ANOTHER_SECOND_LEVEL_SUBTYPE is SAMPLE_FIRST_LEVEL_SUBTYPE;
```

```
  type SAMPLE_SECOND_LEVEL_RECORD is
    record
      FIRST_LEVEL_RECORD : POSITION_LATLONG;
      SCALAR_2           : OCEAN_NAME;
    end record;

  type SAMPLE_THIRD_LEVEL_RECORD is
    record
      SECOND_LEVEL_RECORD : SAMPLE_SECOND_LEVEL_RECORD;
      FIRST_LEVEL_RECORD  : SHIP_CREW;
      SCALAR_3            : CREW_COUNT;
    end record;

end BOATS;
--------------------------------------------------------------------------------
TABLE OCEAN

FIELD NAME STRING 16
FIELD ANALYST STRING 20

TABLE ANALYST

FIELD NAME STRING 20
FIELD SALARY FLOAT 7
FIELD MANAGER STRING 20

TABLE SHIP

FIELD NAME STRING 15
FIELD OCEAN STRING 16
FIELD LAT STRING 7
FIELD LONG STRING 8
FIELD TY_PE STRING 10

TABLE CREW

FIELD TY_PE STRING 10
FIELD SPECIALTY STRING 21
FIELD NUMBER INTEGER 6

TABLE SAMPLE_THIRD_LEVEL_RECORD

FIELD LAT STRING 7
FIELD LONG STRING 8
FIELD SCALAR_2 STRING 16
FIELD TY_PE STRING 10
FIELD SPECIALTY STRING 21
FIELD SCALAR_3 INTEGER 6

END
--------------------------------------------------------------------------------
  DEFINE_TABLE("OCEAN",
   "NAME (INDIAN,ATLANTIC,PACIFIC,MEDITERRANEAN,ARCTIC,CARRIBEAN," &
     "SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);" &
   "ANALYST STRING 1..20");

  DEFINE_TABLE("ANALYST",
```

```
      "NAME STRING 1..20;" &
      "SALARY FLOAT;" &
      "MANAGER STRING 1..20");

    DEFINE_TABLE("SHIP",
      "NAME STRING 1..15;" &
      "OCEAN (INDIAN,ATLANTIC,PACIFIC,MEDITERRANEAN,ARCTIC,CARRIBEAN," &
        "SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);" &
      "LATLONG " &
        "LAT STRING 1..7," &
        "LONG STRING 1..8;" &
      "TY_PE (CARRIER,DESTROYER)");

    DEFINE_TABLE("CREW",
      "KEY " &
        "TY_PE (CARRIER,DESTROYER)," &
        "SPECIALTY (COOK,SHUFFLEBOARD_TEACHER);" &
      "NUMBER INTEGER");

    DEFINE_TABLE("SAMPLE_THIRD_LEVEL_RECORD",
      "SECOND_LEVEL_RECORD " &
        "LAT STRING 1..7," &
        "LONG STRING 1..8," &
        "SCALAR_2 (INDIAN,ATLANTIC,PACIFIC,MEDITERRANEAN,ARCTIC,CARRIBEAN," &
          "SOUTH_CHINA,BERING,GULF_OF_MEXICO,HUDSON_BAY);" &
      "FIRST_LEVEL_RECORD " &
        "TY_PE (CARRIER,DESTROYER)," &
        "SPECIALTY (COOK,SHUFFLEBOARD_TEACHER);" &
      "SCALAR_3 INTEGER");

-)
```

II.3. Ada/SQL DML Prototype Implementation Software

This section contains a program illustrating the use of the SQL SELECT, UPDATE, DELETE, and INSERT functions in Ada. A prototype database management system (PDBMS) loads a database descriptor (data dictionary) and database contents, then processes the data structures built by the Ada/SQL functions to actually perform DML operations.

Input file: DATE.DAT - description and contents of a database taken from Chris Date's book Database: A Primer.

Output file: DML.OUT - script of screen output when the program was run using the Data General (Rolm) validated Ada compiler. Shows (1) results of DML operations that have been implemented in the PDBMS and (2) pretty print of sample complex queries that can be built using Ada/SQL data structures, but for which functionality has not yet been implemented in the PDBMS.

Notes on the main program: CREATE_LINE and SET_LINE manage the output file. LOAD_DATABASE causes the PDBMS to read a database descriptor and contents file. SET_DATABASE causes that database to be used for subsequent Ada/SQL operations. EXECUTE causes the PDBMS to perform the Ada/SQL operation indicated by its parameter. For retrievals, successive records are retrieved using NEXT_RECORD, and FETCH is used to retrieve individual field values within a record. SHOW pretty prints, in SQL notation, the Ada/SQL operation indicated by its parameter. Note that queries can be built and SHOWn for the full range of SQL, indicating that Ada/SQL functions and data structures can successfully implement all of SQL. Not all queries can currently be executed by the PDBMS, however, as full SQL functionality has not yet been implemented for it.

Source files (.ADA) in compilation order:

```
Filename  Package name       Description
```

--------	------------	-----------
TXTPRT	TEXT_PRINT	Manage output file, including continuation lines. Also provide minimum width and default format printing of numbers. Same as TEXT_PRINT for DDL, except without phantoms
TXTINP	TEXT_INPUT	Manage input file, simplify reading input as tokens
SQLDEF	SQL_DEFINITIONS	Defines data structures, generic and other functions
SQLOPS	SQL_OPERATIONS	Defines all Ada/SQL operations
DATEUND	DATE_UNDERLYING	Defines (mostly) data structures for Date's database
DATEDB	DATE_DATABASE	Defines functions used to access Date's database
PGMFUNC	PROGRAM_FUNCTIONS	Execution of data manipulation operations
BULKFUNC	BULK_FUNCTIONS	Database load and save
SHOW	SHOW_PACKAGE	Pretty print formatted SQL from Ada/SQL data structures
MAIN	MAIN	Main program, contains examples of Ada/SQL DML

```
with TEXT_IO;
  use TEXT_IO;

package TEXT_PRINT is

  type LINE_TYPE is limited private;

  type BREAK_TYPE is (BREAK, NO_BREAK);

  type PHANTOM_TYPE is private;

  procedure CREATE_LINE(LINE : in out LINE_TYPE; LENGTH : in POSITIVE);

  procedure SET_LINE(LINE : in LINE_TYPE);

  function CURRENT_LINE return LINE_TYPE;

  procedure SET_INDENT(LINE   : in LINE_TYPE; INDENT : in NATURAL);
  procedure SET_INDENT(INDENT : in NATURAL);

  procedure SET_CONTINUATION_INDENT(LINE   : in LINE_TYPE;
                                    INDENT : in INTEGER);
  procedure SET_CONTINUATION_INDENT(INDENT : in INTEGER);

  function MAKE_PHANTOM(S : STRING) return PHANTOM_TYPE;

  procedure SET_PHANTOMS(LINE          : in LINE_TYPE;
                         START_PHANTOM,
                         END_PHANTOM   : in PHANTOM_TYPE);
```

```
procedure SET_PHANTOMS(START_PHANTOM, END_PHANTOM : in PHANTOM_TYPE);

procedure PRINT(FILE : in FILE_TYPE;
                LINE : in LINE_TYPE;
                ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK);
procedure PRINT(FILE : in FILE_TYPE;
                ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK);
procedure PRINT(LINE : in LINE_TYPE;
                ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK);
procedure PRINT(ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK);

procedure PRINT_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE);
procedure PRINT_LINE(FILE : in FILE_TYPE);
procedure PRINT_LINE(LINE : in LINE_TYPE);
procedure PRINT_LINE;

procedure BLANK_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE);
procedure BLANK_LINE(FILE : in FILE_TYPE);
procedure BLANK_LINE(LINE : in LINE_TYPE);
procedure BLANK_LINE;

generic
  type NUM is range <>;
package INTEGER_PRINT is

  procedure PRINT(FILE : in FILE_TYPE;
                  LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);

  procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM);

end INTEGER_PRINT;

generic
  type NUM is digits <>;
package FLOAT_PRINT is

  procedure PRINT(FILE : in FILE_TYPE;
                  LINE : in LINE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
  procedure PRINT(FILE : in FILE_TYPE;
                  ITEM : in NUM;
                  BRK  : in BREAK_TYPE := BREAK);
```

```
      procedure PRINT(LINE : in LINE_TYPE;
                      ITEM : in NUM;
                      BRK  : in BREAK_TYPE := BREAK);
      procedure PRINT(ITEM : in NUM;
                      BRK  : in BREAK_TYPE := BREAK);

      procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM);

    end FLOAT_PRINT;

    NULL_PHANTOM : constant PHANTOM_TYPE;

    LAYOUT_ERROR : exception renames TEXT_IO.LAYOUT_ERROR;

  private

    type PHANTOM_TYPE is access STRING;

    type LINE_REC(LENGTH : INTEGER) is
      record
        USED_YET            : BOOLEAN := FALSE;
        INDENT              : INTEGER := 0;
        CONTINUATION_INDENT : INTEGER := 2;
        BREAK               : INTEGER := 1;
        INDEX               : INTEGER := 1;
        DATA                : STRING(1..LENGTH);
        START_PHANTOM,
        END_PHANTOM         : PHANTOM_TYPE := NULL_PHANTOM;
      end record;

    type LINE_TYPE is access LINE_REC;

    NULL_PHANTOM : constant PHANTOM_TYPE := new STRING'("");

  end TEXT_PRINT;

  package body TEXT_PRINT is

    DEFAULT_LINE : LINE_TYPE;

    procedure CREATE_LINE(LINE : in out LINE_TYPE; LENGTH : in POSITIVE) is
    begin
      LINE := new LINE_REC(LENGTH);
    end CREATE_LINE;

    procedure SET_LINE(LINE : in LINE_TYPE) is
    begin
      DEFAULT_LINE := LINE;
    end SET_LINE;

    function CURRENT_LINE return LINE_TYPE is
    begin
      return DEFAULT_LINE;
    end CURRENT_LINE;

    procedure SET_INDENT(LINE   : in LINE_TYPE; INDENT : in NATURAL) is
    begin
```

```
  if INDENT >= LINE.LENGTH then
    raise LAYOUT_ERROR;
  end if;
  if LINE.INDEX = LINE.INDENT + 1 then
    for I in 1..INDENT loop
      LINE.DATA(I) := ' ';
    end loop;
    LINE.INDEX := INDENT + 1;
  end if;
  LINE.INDENT := INDENT;
end SET_INDENT;

procedure SET_INDENT(INDENT : in NATURAL) is
begin
  SET_INDENT(DEFAULT_LINE, INDENT);
end SET_INDENT;

procedure SET_CONTINUATION_INDENT(LINE   : in LINE_TYPE;
                                  INDENT : in INTEGER) is
begin
  if LINE.INDENT + INDENT >= LINE.LENGTH or else LINE.INDENT + INDENT < 0
      then
    raise LAYOUT_ERROR;
  end if;
  LINE.CONTINUATION_INDENT := INDENT;
end SET_CONTINUATION_INDENT;

procedure SET_CONTINUATION_INDENT(INDENT : in INTEGER) is
begin
  SET_CONTINUATION_INDENT(DEFAULT_LINE, INDENT);
end SET_CONTINUATION_INDENT;

function MAKE_PHANTOM(S : STRING) return PHANTOM_TYPE is
begin
  return new STRING'(S);
end MAKE_PHANTOM;

procedure SET_PHANTOMS(LINE          : in LINE_TYPE;
                       START_PHANTOM,
                       END_PHANTOM   : in PHANTOM_TYPE) is
begin
  LINE.START_PHANTOM := START_PHANTOM;
  LINE.END_PHANTOM := END_PHANTOM;
end SET_PHANTOMS;

procedure SET_PHANTOMS(START_PHANTOM, END_PHANTOM : in PHANTOM_TYPE) is
begin
  SET_PHANTOMS(DEFAULT_LINE, START_PHANTOM, END_PHANTOM);
end SET_PHANTOMS;

procedure PRINT(FILE : in FILE_TYPE;
                LINE : in LINE_TYPE;
                ITEM : in STRING;
                BRK  : in BREAK_TYPE := BREAK) is
  NEW_BREAK, NEW_INDEX : INTEGER;
begin
  if LINE.INDEX + ITEM'LENGTH + LINE.END_PHANTOM'LENGTH > LINE.LENGTH + 1
```

```
        then
      if LINE.INDENT + LINE.CONTINUATION_INDENT + LINE.START_PHANTOM'LENGTH +
          LINE.INDEX - LINE.BREAK + ITEM'LENGTH > LINE.LENGTH then
        raise LAYOUT_ERROR;
      end if;
      if ITEM = " " and then LINE.END_PHANTOM.all = "" then
        return;
      end if;
      PUT_LINE(FILE,LINE.DATA(1..LINE.BREAK-1) & LINE.END_PHANTOM.all);
      for I in 1..LINE.INDENT + LINE.CONTINUATION_INDENT loop
        LINE.DATA(I) := ' ';
      end loop;
      NEW_BREAK := LINE.INDENT + LINE.CONTINUATION_INDENT + 1;
      NEW_INDEX := NEW_BREAK + LINE.START_PHANTOM'LENGTH +
          LINE.INDEX - LINE.BREAK;
      LINE.DATA(NEW_BREAK..NEW_INDEX) := LINE.START_PHANTOM.all &
          LINE DATA(LINE.BREAK..LINE.INDEX);
      LINE.BREAK := NEW_BREAK;
      LINE.INDEX  = NEW_INDEX;
    end if;
    NEW_INDEX := LINE.INDEX + ITEM'LENGTH;
    LINE.DATA(LINE INDEX..NEW_INDEX-1) := ITEM;
    LINE INDEX := NEW_INDEX;
    if BRK = BREAK then
      LINE.BREAK := NEW_INDEX;
    end if;
    LINE.USED_YET := TRUE;
  end PRINT;

  procedure PRINT(FILE : in FILE_TYPE;
                  ITEM : in STRING;
                  BRK  : in BREAK_TYPE := BREAK) is
  begin
    PRINT(FILE,DEFAULT_LINE,ITEM,BRK);
  end PRINT;

  procedure PRINT(LINE : in LINE_TYPE;
                  ITEM : in STRING;
                  BRK  : in BREAK_TYPE := BREAK) is
  begin
    PRINT(CURRENT_OUTPUT,LINE,ITEM,BRK);
  end PRINT;

  procedure PRINT(ITEM : in STRING; BRK : in BREAK_TYPE := BREAK) is
  begin
    PRINT(CURRENT_OUTPUT,DEFAULT_LINE,ITEM,BRK);
  end PRINT;

  procedure PRINT_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE) is
  begin
    if LINE.INDEX /= LINE.INDENT + 1 then
      PUT_LINE(FILE,LINE.DATA(1..LINE.INDEX-1));
    end if;
    for I in 1..LINE.INDENT loop
      LINE.DATA(I) := ' ';
    end loop;
    LINE.INDEX := LINE.INDENT + 1;
```

```
    LINE.BREAK := LINE.INDEX;
  end PRINT_LINE;

  procedure PRINT_LINE(FILE : in FILE_TYPE) is
  begin
    PRINT_LINE(FILE,DEFAULT_LINE);
  end PRINT_LINE;

  procedure PRINT_LINE(LINE : in LINE_TYPE) is
  begin
    PRINT_LINE(CURRENT_OUTPUT,LINE);
  end PRINT_LINE;

  procedure PRINT_LINE is
  begin
    PRINT_LINE(CURRENT_OUTPUT,DEFAULT_LINE);
  end PRINT_LINE;

  procedure BLANK_LINE(FILE : in FILE_TYPE; LINE : in LINE_TYPE) is
  begin
    if LINE.USED_YET then
      NEW_LINE(FILE);
    end if;
  end BLANK_LINE;

  procedure BLANK_LINE(FILE : in FILE_TYPE) is
  begin
    BLANK_LINE(FILE,DEFAULT_LINE);
  end BLANK_LINE;

  procedure BLANK_LINE(LINE : in LINE_TYPE) is
  begin
    BLANK_LINE(CURRENT_OUTPUT,LINE);
  end BLANK_LINE;

  procedure BLANK_LINE is
  begin
    BLANK_LINE(CURRENT_OUTPUT,DEFAULT_LINE);
  end BLANK_LINE;

  package body INTEGER_PRINT is

    procedure PRINT(FILE : in FILE_TYPE;
                    LINE : in LINE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
      S : STRING(1..NUM'WIDTH);
      L : NATURAL;
    begin
      PRINT(S,L,ITEM);
      PRINT(FILE,LINE,S(1..L),BRK);
    end PRINT;

    procedure PRINT(FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
    begin
```

```
      PRINT(FILE,DEFAULT_LINE,ITEM,BRK);
    end PRINT;

    procedure PRINT(LINE : in LINE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
    begin
      PRINT(CURRENT_OUTPUT,LINE,ITEM,BRK);
    end PRINT;

    procedure PRINT(ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
    begin
      PRINT(CURRENT_OUTPUT,DEFAULT_LINE,ITEM,BRK);
    end PRINT;

    procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM) is
      S : constant STRING := NUM'IMAGE(ITEM);
      F : NATURAL := S'FIRST; -- Bug in DG Compiler -- S'FIRST /= 1 ! ! ! ! ! !
      L : NATURAL;
    begin
      if S(F) = ' ' then
        F := F + 1;
      end if;
      if TO'LENGTH < S'LAST - F + 1 then
        raise LAYOUT_ERROR;
      end if;
      L := TO'FIRST + S'LAST - F;
      TO(TO'FIRST..L) := S(F..S'LAST);
      LAST := L;
    end PRINT;

  end INTEGER_PRINT;

  package body FLOAT_PRINT is

    package NUM_IO is new FLOAT_IO(NUM);
      use NUM_IO;

    procedure PRINT(FILE : in FILE_TYPE;
                    LINE : in LINE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
      S : STRING(1..DEFAULT_FORE + DEFAULT_AFT + DEFAULT_EXP + 2);
      L : NATURAL;
    begin
      PRINT(S,L,ITEM);
      PRINT(FILE,LINE,S(1..L),BRK);
    end PRINT;

    procedure PRINT(FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    BRK  : in BREAK_TYPE := BREAK) is
    begin
      PRINT(FILE,DEFAULT_LINE,ITEM,BRK);
    end PRINT;
```

```
procedure PRINT(LINE : in LINE_TYPE;
                ITEM : in NUM;
                BRK  : in BREAK_TYPE := BREAK) is
begin
  PRINT(CURRENT_OUTPUT, LINE, ITEM, BRK);
end PRINT;

procedure PRINT(ITEM : in NUM;
                BRK  : in BREAK_TYPE := BREAK) is
begin
  PRINT(CURRENT_OUTPUT, DEFAULT_LINE, ITEM, BRK);
end PRINT;

procedure PRINT(TO : out STRING; LAST : out NATURAL; ITEM : in NUM) is
  S         : STRING(1..DEFAULT_FORE + DEFAULT_AFT + DEFAULT_EXP + 2);
  EXP       : INTEGER;
  E_INDEX   : NATURAL := S'LAST - DEFAULT_EXP;
  DOT_INDEX : NATURAL := DEFAULT_FORE + 1;
  L         : NATURAL := 0;
begin
  PUT(S, ITEM);
  EXP := INTEGER'VALUE(S(E_INDEX+1..S'LAST));
  if EXP >= 0 and then EXP <= DEFAULT_AFT-1 then
    S(DOT_INDEX..DOT_INDEX+EXP-1) := S(DOT_INDEX+1..DOT_INDEX+EXP);
    S(DOT_INDEX+EXP) := '.';
    for I in E_INDEX..S'LAST loop
      S(I) := ' ';
    end loop;
  end if;
  for I in reverse 1..E_INDEX-1 loop
    exit when S(I) /= '0' or else S(I-1) = '.';
    S(I) := ' ';
  end loop;
  for I in S'RANGE loop
    if S(I) /= ' ' then
      L := L + 1;
      TO(L) := S(I);
    end if;
  end loop;
  LAST := L;
exception
  when CONSTRAINT_ERROR =>
    raise LAYOUT_ERROR;
end PRINT;

end FLOAT_PRINT;

end TEXT_PRINT;
```

```
with TEXT_IO;
  use TEXT_IO;

package TEXT_INPUT is

  type STRING_LINK is access STRING;

  type BUFFER_TYPE is private;

  package INTEGER_IO is new TEXT_IO.INTEGER_IO(INTEGER);
  package FLOAT_IO   is new TEXT_IO.FLOAT_IO(FLOAT);
    use INTEGER_IO, FLOAT_IO;

  function MAKE_BUFFER(LENGTH : POSITIVE) return BUFFER_TYPE;

  procedure OPEN_INPUT(BUFFER : in out BUFFER_TYPE;
                       MODE   : in FILE_MODE;
                       NAME   : in STRING);

  procedure CLOSE_INPUT(BUFFER : in out BUFFER_TYPE);

  function END_OF_FILE(BUFFER : BUFFER_TYPE) return BOOLEAN;

  procedure CARD_ERROR(BUFFER : in BUFFER_TYPE; MESSAGE : in STRING);

  procedure IN_IDENT(BUFFER : in out BUFFER_TYPE; -- calls NEXT_TOKEN!
                     IDENT  : out    STRING;      -- leaves ptr after ident
                     LAST   : out    NATURAL);

  function IN_INTEGER(BUFFER : BUFFER_TYPE) return INTEGER;
  function IN_FLOAT  (BUFFER : BUFFER_TYPE) return FLOAT;
  function IN_STRING (BUFFER : BUFFER_TYPE) return STRING_LINK;

private

  type BUFFER_REC(LENGTH : POSITIVE) is
    record
      BUFFER : STRING(1..LENGTH);
      FILE   : FILE_TYPE;
      NEXT   : POSITIVE := 1;
      LAST   : NATURAL  := 0;
    end record;

  type BUFFER_TYPE is access BUFFER_REC;

end TEXT_INPUT;

with TEXT_IO;
  use TEXT_IO;

package body TEXT_INPUT is

  function MAKE_BUFFER(LENGTH : POSITIVE) return BUFFER_TYPE is
  begin
    return new BUFFER_REC(LENGTH);
  end MAKE_BUFFER;
```

```
procedure OPEN_INPUT(BUFFER : in out BUFFER_TYPE;
                     MODE   : in FILE_MODE;
                     NAME   : in STRING) is
begin
  OPEN(BUFFER.FILE,MODE,NAME);
end OPEN_INPUT;

procedure CLOSE_INPUT(BUFFER : in out BUFFER_TYPE) is
begin
  CLOSE(BUFFER.FILE);
end CLOSE_INPUT;

function END_OF_FILE(BUFFER : BUFFER_TYPE) return BOOLEAN is
begin
  return END_OF_FILE(BUFFER.FILE);
end END_OF_FILE;

procedure CARD_ERROR(BUFFER : in BUFFER_TYPE; MESSAGE : in STRING) is
begin
  PUT_LINE("***** Error on input card:");
  PUT_LINE(BUFFER.BUFFER(1..BUFFER.LAST));
  PUT_LINE(MESSAGE);
  raise DATA_ERROR;
end CARD_ERROR;

procedure NEXT_LINE(BUFFER : in BUFFER_TYPE) is
begin
  loop
    GET_LINE(BUFFER.FILE,BUFFER.BUFFER,BUFFER.LAST);
    exit when BUFFER.LAST >= 2 and then BUFFER.BUFFER(1..2) /= "--";
    exit when BUFFER.LAST = 1;
  end loop;
  BUFFER.NEXT := 1;
end NEXT_LINE;

procedure NEXT_TOKEN(BUFFER : in BUFFER_TYPE) is
begin
  loop
    if BUFFER.NEXT > BUFFER.LAST then
      NEXT_LINE(BUFFER);
    end if;
    if BUFFER.BUFFER(BUFFER.NEXT) = '-' and then
        BUFFER.NEXT < BUFFER.LAST and then
        BUFFER.BUFFER(BUFFER.NEXT+1) = '-' then
      NEXT_LINE(BUFFER);
    end if;
    exit when BUFFER.BUFFER(BUFFER.NEXT) /= ' ' and then
        BUFFER.BUFFER(BUFFER.NEXT) /= ASCII.HT;
    BUFFER.NEXT := BUFFER.NEXT + 1;
  end loop;
end NEXT_TOKEN;

function TOKEN_END(BUFFER : BUFFER_TYPE) return POSITIVE is
  PTR : POSITIVE;
begin
  NEXT_TOKEN(BUFFER);
  PTR := BUFFER.NEXT;
```

```
    while PTR <= BUFFER.LAST and then BUFFER.BUFFER(PTR) /= ' ' and then
        BUFFER.BUFFER(PTR) /= ASCII.HT loop
      PTR := PTR + 1;
    end loop;
    return PTR-1;
  end TOKEN_END;

  procedure IN_IDENT(BUFFER : in out BUFFER_TYPE;
                     IDENT  : out    STRING;
                     LAST   : out    NATURAL) is
    TOKEND,
    TLAST : POSITIVE;
  begin
    TOKEND := TOKEN_END(BUFFER);
    TLAST := IDENT'FIRST + TOKEND - BUFFER.NEXT;
    IDENT(IDENT'FIRST..TLAST) := BUFFER.BUFFER(BUFFER.NEXT..TOKEND);
    LAST := TLAST;
    BUFFER.NEXT := TOKEND + 1;
  end IN_IDENT;

  function IN_INTEGER(BUFFER : BUFFER_TYPE) return INTEGER is
    TOKEND : POSITIVE;
    INT,
    LAST   : INTEGER;
  begin
    TOKEND := TOKEN_END(BUFFER);
    GET(BUFFER.BUFFER(BUFFER.NEXT..TOKEND),INT,LAST);
    BUFFER.NEXT := TOKEND + 1;
    return INT;
  end IN_INTEGER;

  function IN_FLOAT  (BUFFER : BUFFER_TYPE) return FLOAT is
    TOKEND : POSITIVE;
    FLT    : FLOAT;
    LAST   : INTEGER;
  begin
    TOKEND := TOKEN_END(BUFFER);
    GET(BUFFER.BUFFER(BUFFER.NEXT..TOKEND),FLT,LAST);
    BUFFER.NEXT := TOKEND + 1;
    return FLT;
  end IN_FLOAT;

  function IN_STRING (BUFFER : BUFFER_TYPE) return STRING_LINK is
    PTR : POSITIVE;
    STR : STRING_LINK;
  begin
    NEXT_TOKEN(BUFFER);
    if BUFFER.BUFFER(BUFFER.NEXT) /= '"' then
      raise DATA_ERROR;
    end if;
    PTR := BUFFER.NEXT + 1;
    while PTR <= BUFFER.LAST and then BUFFER.BUFFER(PTR) /= '"' loop
      PTR := PTR + 1;
    end loop;
    if PTR > BUFFER.LAST then
      raise DATA_ERROR;
    end if;
```

```
      STR := new STRING(1..PTR-BUFFER.NEXT-1);
      STR.all := BUFFER.BUFFER(BUFFER.NEXT+1..PTR-1);
      BUFFER.NEXT := PTR + 1;
      return STR;
    end IN_STRING;

  end TEXT_INPUT;
```

```
with TEXT_INPUT;
  use TEXT_INPUT;

package SQL_DEFINITIONS is

  type TABLE is private;
  type FIELD is private;

  type TABLE_NAME is private;
  type FIELD_NAME is private;

  subtype STRING_LINK is TEXT_INPUT.STRING_LINK;

  type OPERATOR_TYPE is (O_SELECT, O_INSERT, O_DELETE, O_UPDATE, O_LIKE,
      O_SUM, O_AVG, O_MAX, O_MIN, O_COUNT, O_IN, O_EXISTS, O_DESC, O_AND,
      O_OR, O_XOR, O_EQ, O_NE, O_LT, O_LE, O_GT, O_GE, O_PLUS, O_MINUS, O_CAT,
      O_UNARY_PLUS, O_UNARY_MINUS, O_TIMES, O_DIV, O_MOD, O_REM, O_POWER,
      O_ABS, O_NOT);

  STAR,
  NULL_FIELD : constant FIELD;
  NULL_TABLE : constant TABLE;

  function MAKE_TABLE_NAME(NAME : STRING) return TABLE_NAME;

  function MAKE_FIELD(RELATION : TABLE_NAME; TEMPLATE : FIELD) return FIELD;

  function MAKE_FIELD(NAME : STRING) return FIELD;

  function TABLEIFY(F : FIELD) return TABLE;

  function FIELDIFY(F : FIELD)      return FIELD;
  function FIELDIFY(F : INTEGER)    return FIELD;
  function FIELDIFY(F : FLOAT)      return FIELD;
  function FIELDIFY(F : STRING)     return FIELD;

  function L_FIELDIFY(F : FIELD)      return FIELD renames FIELDIFY;
  function L_FIELDIFY(F : INTEGER)    return FIELD renames FIELDIFY;
  function L_FIELDIFY(F : FLOAT)      return FIELD renames FIELDIFY;
  function L_FIELDIFY(F : STRING)     return FIELD renames FIELDIFY;

  function R_FIELDIFY(F : FIELD)      return FIELD renames FIELDIFY;
  function R_FIELDIFY(F : INTEGER)    return FIELD renames FIELDIFY;
  function R_FIELDIFY(F : FLOAT)      return FIELD renames FIELDIFY;
  function R_FIELDIFY(F : STRING)     return FIELD renames FIELDIFY;

  generic
    TABLE_FIELD : FIELD;
  function GET_TABLE return TABLE;

  generic
    FIELD_NAME : FIELD;
  function GET_FIELD_NAME return FIELD;

  generic
    type TABLE_TYPE is private;
    DATA : TABLE_TYPE;
```

```
function GET_FIELDS return TABLE_TYPE;

generic
  TABLE_FIELD : FIELD;
function INSERT_FIELDS(F : in FIELD) return FIELD;

generic
  type VALUE_TYPE is private;
  with function FIELDIFY(F : VALUE_TYPE) return FIELD is <>;
function VALUES_GEN(V : VALUE_TYPE) return FIELD;

generic
  OPCODE : OPERATOR_TYPE;
  type L_TYPE is private;
  with function L_FIELDIFY(F : L_TYPE) return FIELD is <>;
function UNARY_OPERATOR(L : L_TYPE) return FIELD;

generic
  OPCODE : OPERATOR_TYPE;
  type L_TYPE is private;
  type R_TYPE is private;
  with function L_FIELDIFY(F : L_TYPE) return FIELD is <>;
  with function R_FIELDIFY(F : R_TYPE) return FIELD is <>;
function BINARY_OPERATOR(L : L_TYPE; R : R_TYPE) return FIELD;

function SELEC(WHAT   : FIELD := NULL_FIELD;
               FROM   : TABLE := NULL_TABLE;
               WHERE  : FIELD := NULL_FIELD;
               GROUP  : FIELD := NULL_FIELD;
               HAVING : FIELD := NULL_FIELD;
               ORDER  : FIELD := NULL_FIELD) return FIELD;

function INSERT_INTO(WHAT   : FIELD;
                     VALUES : FIELD) return FIELD;

function INSERT_INTO(WHAT   : TABLE;
                     VALUES : FIELD) return FIELD;

function INSERT_UNTO(WHAT   : FIELD;
                     VALUES : FIELD) return FIELD renames INSERT_INTO;

function INSERT_UNTO(WHAT   : TABLE;
                     VALUES : FIELD) return FIELD renames INSERT_INTO;

generic
  type WHAT_TYPE is private;
  type VALUE_TYPE is private;
  with function INSERT_UNTO(WHAT : WHAT_TYPE; VALUES: FIELD) return FIELD
      is <>;
  with function FIELDIFY(VALUE : VALUE_TYPE) return FIELD is <>;
function INSERT_GEN(WHAT : WHAT_TYPE; VALUES : VALUE_TYPE) return FIELD;

function DELETE(FROM  : TABLE := NULL_TABLE;
                WHERE : FIELD := NULL_FIELD) return FIELD;

function UPDATE(WHAT  : TABLE := NULL_TABLE;
                SET   : FIELD;
```

```
                    WHERE : FIELD := NULL_FIELD) return FIELD;

function "&"(L : TABLE; R : TABLE) return TABLE;

package SQL_FUNCTIONS is

  type DATABASE_TYPE is private;
  type VALUE_LINK    is private;
  type RECORD_LINK   is private;

  type EXTENDED_FIELD_INDEX is new NATURAL;
  subtype FIELD_INDEX is EXTENDED_FIELD_INDEX
      range 1..EXTENDED_FIELD_INDEX'LAST;

  type EXTENDED_TABLE_INDEX is new NATURAL;
  subtype TABLE_INDEX is EXTENDED_TABLE_INDEX
      range 1..EXTENDED_TABLE_INDEX'LAST;

  package PROGRAM_FUNCTIONS is

    type CURSOR_TYPE is private;

    function  EXECUTE     (F : in FIELD) return CURSOR_TYPE;
    procedure EXECUTE     (F : in FIELD);
    procedure LIST        (F : in FIELD);
    procedure SET_DATABASE(DB : in DATABASE_TYPE);
    procedure NEXT_RECORD (CURSOR : in out CURSOR_TYPE);

    procedure FETCH(CURSOR : in  CURSOR_TYPE;
                    FIELD  : in  FIELD_INDEX;
                    INT    : out INTEGER);

    procedure FETCH(CURSOR : in  CURSOR_TYPE;
                    FIELD  : in  FIELD_INDEX;
                    FLT    : out FLOAT);

    procedure FETCH(CURSOR : in  CURSOR_TYPE;
                    FIELD  : in  FIELD_INDEX;
                    STR    : out STRING;
                    LAST   : out NATURAL);

    function FETCH(CURSOR : CURSOR_TYPE;
                   FIELD  : FIELD_INDEX) return INTEGER;

    function FETCH(CURSOR : CURSOR_TYPE;
                   FIELD  : FIELD_INDEX) return FLOAT;

    function FETCH(CURSOR : CURSOR_TYPE;
                   FIELD  : FIELD_INDEX) return STRING;

    CALL_ERROR          : exception;
    DONE_ERROR          : exception;
    FIELD_ERROR         : exception;
    SYNTAX_ERROR        : exception;
    TABLE_ERROR         : exception;
    TRUNCATE_ERROR      : exception;
    TYPE_ERROR          : exception;
```

```
  UNIMPLEMENTED_ERROR : exception;

private

  type QUERY_NODE_REC;

  type QUERY_NODE is access QUERY_NODE_REC;

  type QUERY_NODE_REC is
    record
      NEXT_NODE : QUERY_NODE;
      FIELD     : FIELD_INDEX;
      VALUE     : VALUE_LINK;
    end record;

  type CURSOR_TYPE is
    record
      QUERY          : QUERY_NODE;
      CURRENT_RECORD : RECORD_LINK;
      NEW_QUERY      : BOOLEAN := TRUE;
    end record;

end PROGRAM_FUNCTIONS;

package SHOW_PACKAGE is
  procedure SHOW (F : in FIELD);
end SHOW_PACKAGE;

package BULK_FUNCTIONS is
  function  LOAD_DATABASE(FILE_NAME : in STRING) return DATABASE_TYPE;
  procedure SAVE_DATABASE(FILE_NAME : in STRING;
                          DATABASE  : in DATABASE_TYPE);
end BULK_FUNCTIONS;

subtype CURSOR_TYPE is PROGRAM_FUNCTIONS.CURSOR_TYPE;

function EXECUTE(F : FIELD) return CURSOR_TYPE
    renames PROGRAM_FUNCTIONS.EXECUTE;

procedure EXECUTE(F : in FIELD) renames PROGRAM_FUNCTIONS.EXECUTE;
procedure LIST   (F : in FIELD) renames PROGRAM_FUNCTIONS.LIST;
procedure SHOW   (F : in FIELD) renames SHOW_PACKAGE.SHOW;

procedure SET_DATABASE(DB : in DATABASE_TYPE)
    renames PROGRAM_FUNCTIONS.SET_DATABASE;

procedure NEXT_RECORD(CURSOR : in out CURSOR_TYPE)
    renames PROGRAM_FUNCTIONS.NEXT_RECORD;

procedure FETCH(CURSOR : in  CURSOR_TYPE;
                FIELD  : in  FIELD_INDEX;
                INT    : out INTEGER) renames PROGRAM_FUNCTIONS.FETCH;

procedure FETCH(CURSOR : in  CURSOR_TYPE;
                FIELD  : in  FIELD_INDEX;
                FLT    : out FLOAT) renames PROGRAM_FUNCTIONS.FETCH;
```

```
    procedure FETCH(CURSOR : in  CURSOR_TYPE;
                    FIELD  : in  FIELD_INDEX;
                    STR    : out STRING;
                    LAST   : out NATURAL) renames PROGRAM_FUNCTIONS.FETCH;

    function FETCH(CURSOR : CURSOR_TYPE;
mes PROGRAM_FUNCTIONS.FETCH;

    function FETCH(CURSOR : CURSOR_TYPE;
                   FIELD  : FIELD_INDEX) return FLOAT
        renames PROGRAM_FUNCTIONS.FETCH;

    function FETCH(CURSOR : CURSOR_TYPE;
                   FIELD  : FIELD_INDEX) return STRING
        renames PROGRAM_FUNCTIONS.FETCH;

    function LOAD_DATABASE(FILE_NAME : in STRING) return DATABASE_TYPE renames
      BULK_FUNCTIONS.LOAD_DATABASE;

    procedure SAVE_DATABASE(FILE_NAME : in STRING;
                            DATABASE  : in DATABASE_TYPE) renames
      BULK_FUNCTIONS.SAVE_DATABASE;

    CALL_ERROR          : exception renames PROGRAM_FUNCTIONS.CALL_ERROR;
    DONE_ERROR          : exception renames PROGRAM_FUNCTIONS.DONE_ERROR;
    FIELD_ERROR         : exception renames PROGRAM_FUNCTIONS.FIELD_ERROR;
    SYNTAX_ERROR        : exception renames PROGRAM_FUNCTIONS.SYNTAX_ERROR;
    TABLE_ERROR         : exception renames PROGRAM_FUNCTIONS.TABLE_ERROR;
    TRUNCATE_ERROR      : exception renames PROGRAM_FUNCTIONS.TRUNCATE_ERROR;
    TYPE_ERROR          : exception renames PROGRAM_FUNCTIONS.TYPE_ERROR;
    UNIMPLEMENTED_ERROR : exception renames
        PROGRAM_FUNCTIONS.UNIMPLEMENTED_ERROR;

  private

    type DATABASE_FIELD_TYPE is (INTEGER_FIELD, FLOAT_FIELD, STRING_FIELD);

    type VALUE_TYPE(FIELD_TYPE : DATABASE_FIELD_TYPE) is
      record
        case FIELD_TYPE is
          when INTEGER_FIELD =>
            INTEGER_VALUE : INTEGER;
          when FLOAT_FIELD =>
            FLOAT_VALUE   : FLOAT;
          when STRING_FIELD =>
            STRING_VALUE  : STRING_LINK;
        end case;
      end record;

    type VALUE_LINK is access VALUE_TYPE;

    type VALUE_ARRAY is array(FIELD_INDEX range <>) of VALUE_LINK;

    type RECORD_TYPE(NUMBER_FIELDS : EXTENDED_FIELD_INDEX);

    type RECORD_LINK is access RECORD_TYPE;
```

```
  type RECORD_TYPE(NUMBER_FIELDS : EXTENDED_FIELD_INDEX) is
    record
      NEXT_RECORD : RECORD_LINK;
      VALUES      : VALUE_ARRAY(1..NUMBER_FIELDS);
    end record;

  type FIELD_TYPE is
    record
      NAME      : FIELD_NAME;
      DATA_TYPE : DATABASE_FIELD_TYPE;
      SIZE      : POSITIVE;
    end record;

  type FIELD_ARRAY is array(FIELD_INDEX range <>) of FIELD_TYPE;

  type TABLE_TYPE(NUMBER_FIELDS : EXTENDED_FIELD_INDEX) is
    record
      NAME    : TABLE_NAME;
      RECORDS : RECORD_LINK;
      FIELDS  : FIELD_ARRAY(1..NUMBER_FIELDS);
    end record;

  type TABLE_LINK is access TABLE_TYPE;

  type TABLE_ARRAY is array(TABLE_INDEX range <>) of TABLE_LINK;

  type DATABASE_TYPE is access TABLE_ARRAY;

end SQL_FUNCTIONS;

subtype DATABASE_TYPE is SQL_FUNCTIONS.DATABASE_TYPE;
subtype CURSOR_TYPE   is SQL_FUNCTIONS.CURSOR_TYPE;
subtype FIELD_INDEX   is SQL_FUNCTIONS.FIELD_INDEX;

function EXECUTE(F : FIELD) return CURSOR_TYPE
    renames SQL_FUNCTIONS.EXECUTE;

procedure EXECUTE(F : in FIELD) renames SQL_FUNCTIONS.EXECUTE;
procedure LIST   (F : in FIELD) renames SQL_FUNCTIONS.LIST;
procedure SHOW   (F : in FIELD) renames SQL_FUNCTIONS.SHOW;

procedure SET_DATABASE(DB : in DATABASE_TYPE)
    renames SQL_FUNCTIONS.SET_DATABASE;

procedure NEXT_RECORD(CURSOR : in out CURSOR_TYPE)
    renames SQL_FUNCTIONS.NEXT_RECORD;

procedure FETCH(CURSOR : in  CURSOR_TYPE;
                FIELD  : in  FIELD_INDEX;
                INT    : out INTEGER) renames SQL_FUNCTIONS.FETCH;

procedure FETCH(CURSOR : in  CURSOR_TYPE;
                FIELD  : in  FIELD_INDEX;
                FLT    : out FLOAT) renames SQL_FUNCTIONS.FETCH;

procedure FETCH(CURSOR : in  CURSOR_TYPE;
                FIELD  : in  FIELD_INDEX;
```

```
                    STR     : out STRING;
                    LAST    : out NATURAL) renames SQL_FUNCTIONS.FETCH;

  function FETCH(CURSOR : CURSOR_TYPE;
                 FIELD  : FIELD_INDEX) return INTEGER
      renames SQL_FUNCTIONS.FETCH;

  function FETCH(CURSOR : CURSOR_TYPE;
                 FIELD  : FIELD_INDEX) return FLOAT
      renames SQL_FUNCTIONS.FETCH;

  function FETCH(CURSOR : CURSOR_TYPE;
                 FIELD  : FIELD_INDEX) return STRING
      renames SQL_FUNCTIONS.FETCH;

  function  LOAD_DATABASE(FILE_NAME : in STRING) return DATABASE_TYPE renames
      SQL_FUNCTIONS.LOAD_DATABASE;

  procedure SAVE_DATABASE(FILE_NAME : in STRING;
                          DATABASE  : in DATABASE_TYPE) renames
      SQL_FUNCTIONS.SAVE_DATABASE;

  CALL_ERROR           : exception renames SQL_FUNCTIONS.CALL_ERROR;
  DONE_ERROR           : exception renames SQL_FUNCTIONS.DONE_ERROR;
  FIELD_ERROR          : exception renames SQL_FUNCTIONS.FIELD_ERROR;
  SYNTAX_ERROR         : exception renames SQL_FUNCTIONS.SYNTAX_ERROR;
  TABLE_ERROR          : exception renames SQL_FUNCTIONS.TABLE_ERROR;
  TRUNCATE_ERROR       : exception renames SQL_FUNCTIONS.TRUNCATE_ERROR;
  TYPE_ERROR           : exception renames SQL_FUNCTIONS.TYPE_ERROR;
  UNIMPLEMENTED_ERROR  : exception renames SQL_FUNCTIONS.UNIMPLEMENTED_ERROR;

private

  type TABLE_NAME_STRING is new STRING;
  type FIELD_NAME_STRING is new STRING;

  type TABLE_NAME is access TABLE_NAME_STRING;
  type FIELD_NAME is access FIELD_NAME_STRING;

  type TABLE_REC;

  type TABLE is access TABLE_REC;

  type TABLE_REC is
    record
      NAME      : TABLE_NAME;
      NEXT_LINK : TABLE;
    end record;

  type FIELD_TYPE_TYPE is (OPERATOR, INTEGER_LITERAL, STRING_LITERAL
    FLOAT_LITERAL, EMPTY, QUALIFIED_FIELD, UNQUALIFIED_FIELD  FROM_LIST);

  type FIELD_REC(FIELD_TYPE : FIELD_TYPE_TYPE);

  type FIELD is access FIELD_REC;

  type FIELD_REC(FIELD_TYPE : FIELD_TYPE_TYPE) is
```

```
    record
      ACROSS_LINK : FIELD;
      case FIELD_TYPE is
        when FROM_LIST =>
          TABLE_LINK      : TABLE;
        when OPERATOR =>
          OPCODE          : OPERATOR_TYPE;
          DOWN_LINK       : FIELD;
        when INTEGER_LITERAL =>
          INTEGER_VALUE : INTEGER;
        when STRING_LITERAL =>
          STRING_VALUE   : STRING_LINK;
        when FLOAT_LITERAL =>
          FLOAT_VALUE    : FLOAT;
        when EMPTY =>
          null;
        when QUALIFIED_FIELD ! UNQUALIFIED_FIELD =>
          RELATION        : TABLE_NAME; -- null for UNQUALIFIED_FIELD
          NAME            : FIELD_NAME;
      end case;
    end record;

  STAR : constant FIELD := new FIELD_REC'(
    UNQUALIFIED_FIELD,null,null,new FIELD_NAME_STRING'("*"));

  NULL_TABLE : constant TABLE := null;
  NULL_FIELD : constant FIELD := null;

end SQL_DEFINITIONS;

package body SQL_DEFINITIONS is

  function MAKE_TABLE_NAME(NAME   STRING) return TABLE_NAME is
  begin
    return new TABLE_NAME_STRING (TABLE_NAME_STRING(NAME));
  end

  function MAKE_FIELD(RELATION   TABLE_NAME  TEMPLATE   FIELD) return FIELD is
  begin
    return new FIELD_REC (QUALIFIED_FIELD null RELATION TEMPLATE NAME);
  end MAKE_FIELD

  function MAKE_FIELD(NAME   STRING) return FIELD is
  begin
    return new FIELD_REC (
      UNQUALIFIED_FIELD null null
        new FIELD_NAME_STRING (FIELD_NAME_STRING(NAME)) )
  end MAKE_FIELD

  function TABLEIFY(F   FIELD) return TABLE is
  begin
    return new TABLE_REC (F RELATION null)
  end TABLEIFY

  function GET_TABLE return TABLE is
  begin
    return TABLEIFY(TABLE_FIELD)
```

```
end GET_TABLE;

function GET_FIELD_NAME return FIELD is
begin
  return FIELD_NAME;
end GET_FIELD_NAME;

function GET_FIELDS return TABLE_TYPE is
begin
  return DATA;
end GET_FIELDS;

function INSERT_FIELDS(F : in FIELD) return FIELD is
begin
  return new FIELD_REC'(FROM_LIST,F,TABLEIFY(TABLE_FIELD));
end INSERT_FIELDS;

function FIELDIFY(F : FIELD) return FIELD is
begin
  if F = null then
    return new FIELD_REC'(EMPTY,null);
  else
    case F.FIELD_TYPE is
      when QUALIFIED_FIELD | UNQUALIFIED_FIELD =>
        return new FIELD_REC'(F.all);
      when others =>
        return F;
    end case;
  end if;
end FIELDIFY;

function FIELDIFY(F : INTEGER) return FIELD is
begin
  return new FIELD_REC'(INTEGER_LITERAL,null,F);
end FIELDIFY;

function FIELDIFY(F : FLOAT) return FIELD is
begin
  return new FIELD_REC'(FLOAT_LITERAL,null,F);
end FIELDIFY;

function FIELDIFY(F : STRING) return FIELD is
begin
  return new FIELD_REC'(STRING_LITERAL,null,new STRING'(F));
end FIELDIFY;

function VALUES_GEN(V : VALUE_TYPE) return FIELD is
begin
  return FIELDIFY(V);
end VALUES_GEN;

function UNARY_OPERATOR(L : L_TYPE) return FIELD is
begin
  return new FIELD_REC'(OPERATOR,null,OPCODE,L_FIELDIFY(L) );
end UNARY_OPERATOR

function BINARY_OPERATOR(L : L_TYPE; R : R_TYPE) return FIELD is
```

```
    LF : FIELD;
  begin
    LF := L_FIELDIFY(L);
    LF.ACROSS_LINK := R_FIELDIFY(R);
    return new FIELD_REC'(OPERATOR,null,OPCODE,LF);
  end BINARY_OPERATOR;

  function SELEC(WHAT   : FIELD := NULL_FIELD;
                 FROM   : TABLE := NULL_TABLE;
                 WHERE  : FIELD := NULL_FIELD;
                 GROUP  : FIELD := NULL_FIELD;
                 HAVING : FIELD := NULL_FIELD;
                 ORDER  : FIELD := NULL_FIELD) return FIELD is
    RET_VALUE,F : FIELD;
  begin
    F := FIELDIFY(WHAT);
    RET_VALUE := new FIELD_REC'(OPERATOR,null,O_SELECT,F);
    F.ACROSS_LINK := new FIELD_REC'(FROM_LIST,null,FROM); F := F.ACROSS_LINK;
    F.ACROSS_LINK := FIELDIFY(WHERE); F := F.ACROSS_LINK;
    F.ACROSS_LINK := FIELDIFY(GROUP); F := F.ACROSS_LINK;
    F.ACROSS_LINK := FIELDIFY(HAVING); F := F.ACROSS_LINK;
    F.ACROSS_LINK := FIELDIFY(ORDER); F := F.ACROSS_LINK;
    return RET_VALUE;
  end SELEC;

  function INSERT_INTO(WHAT   : FIELD;
                       VALUES : FIELD) return FIELD is
  begin
    return new FIELD_REC'(OPERATOR,FIELDIFY(WHAT),O_INSERT FIELDIFY(VALUES));
  end INSERT_INTO;

  function INSERT_INTO(WHAT   : TABLE;
                       VALUES : FIELD) return FIELD is
  begin
    return new FIELD_REC'(OPERATOR new FIELD_REC'(FROM_LIST null WHAT)
        O_INSERT FIELDIFY(VALUES))
  end INSERT_INTO

  function INSERT_GEN(WHAT  WHAT_TYPE  VALUES  VALUE_TYPE) return FIELD is
  begin
    return INSERT_UNTO(WHAT FIELDIFY(VALUES))
  end INSERT_GEN

  function DELETE(FROM   TABLE  = NULL_TABLE
                  WHERE  FIELD  = NULL_FIELD) return FIELD is
  begin
    return new FIELD_REC (OPERATOR null O_DELETE
        new FIELD_REC (FROM_LIST FIELDIFY(WHERE) FROM)
  end DELETE

  function UPDATE(WHAT   TABLE  = NULL_TABLE
                  SET    FIELD
                  WHERE  FIELD  = NULL_FIELD  return FIELD is
    RET_VALUE  F  FIELD
  begin
    F  = new FIELD_REC (FROM_LIST null WHAT)
    RET_VALUE  = new FIELD_REC (OPERATOR null O_UPDATE F
```

```
    F.ACROSS_LINK := FIELDIFY(SET); F := F.ACROSS_LINK;
    F.ACROSS_LINK := FIELDIFY(WHERE);
    return RET_VALUE;
  end UPDATE;

  function "&"(L : TABLE; R : TABLE) return TABLE is
    LP : TABLE := L;
  begin
    while LP.NEXT_LINK /= null loop
      LP := LP.NEXT_LINK;
    end loop;
    LP.NEXT_LINK := R;
    return L;
  end "&";

  package body SQL_FUNCTIONS is separate;

end SQL_DEFINITIONS;

with TEXT_PRINT;
  use TEXT_PRINT;

separate(SQL_DEFINITIONS)
package body SQL_FUNCTIONS is

  package INT_PRINT is new INTEGER_PRINT(INTEGER);
  package FLT_PRINT is new FLOAT_PRINT(FLOAT);
    use INT_PRINT, FLT_PRINT;

  package body PROGRAM_FUNCTIONS is separate;
  package body SHOW_PACKAGE is separate;
  package body BULK_FUNCTIONS is separate;

end SQL_FUNCTIONS;
```

```
with SQL_DEFINITIONS;
  use SQL_DEFINITIONS;

package SQL_OPERATIONS is

  subtype TABLE is SQL_DEFINITIONS.TABLE;
  subtype FIELD is SQL_DEFINITIONS.FIELD;

  type STAR_TYPE is ('*');

  function SELEC(WHAT   : FIELD := NULL_FIELD;
                 FROM   : TABLE := NULL_TABLE;
                 WHERE  : FIELD := NULL_FIELD;
                 GROUP  : FIELD := NULL_FIELD;
                 HAVING : FIELD := NULL_FIELD;
                 ORDER  : FIELD := NULL_FIELD) return FIELD
    renames SQL_DEFINITIONS.SELEC;

  function SELEC(WHAT   : STAR_TYPE;
                 FROM   : TABLE := NULL_TABLE;
                 WHERE  : FIELD := NULL_FIELD;
                 GROUP  : FIELD := NULL_FIELD;
                 HAVING : FIELD := NULL_FIELD;
                 ORDER  : FIELD := NULL_FIELD) return FIELD;

  function INSERT_INTO(WHAT   : FIELD;
                       VALUES : FIELD) return FIELD
    renames SQL_DEFINITIONS.INSERT_INTO;

  function INSERT_INTO(WHAT   : TABLE;
                       VALUES : FIELD) return FIELD
    renames SQL_DEFINITIONS.INSERT_INTO;

  function INSERT_INTO is new INSERT_GEN(FIELD,INTEGER);
  function INSERT_INTO is new INSERT_GEN(FIELD,FLOAT);
  function INSERT_INTO is new INSERT_GEN(FIELD,STRING);
  function INSERT_INTO is new INSERT_GEN(TABLE,INTEGER);
  function INSERT_INTO is new INSERT_GEN(TABLE,FLOAT);
  function INSERT_INTO is new INSERT_GEN(TABLE,STRING);

  function DELETE(FROM  : TABLE := NULL_TABLE;
                  WHERE : FIELD := NULL_FIELD) return FIELD
    renames SQL_DEFINITIONS.DELETE;

  function UPDATE(WHAT  : TABLE := NULL_TABLE;
                  SET   : FIELD;
                  WHERE : FIELD := NULL_FIELD) return FIELD
    renames SQL_DEFINITIONS.UPDATE;

  function VALUES is new VALUES_GEN(FIELD);
  function VALUES is new VALUES_GEN(INTEGER);
  function VALUES is new VALUES_GEN(FLOAT);
  function VALUES is new VALUES_GEN(STRING);

  function LIKE is new BINARY_OPERATOR(O_LIKE,FIELD, FIELD);
  function LIKE is new BINARY_OPERATOR(O_LIKE,FIELD, STRING);
  function LIKE is new BINARY_OPERATOR(O_LIKE,STRING,FIELD);
```

```
function SUM is new UNARY_OPERATOR(O_SUM,FIELD);
function AVG is new UNARY_OPERATOR(O_AVG,FIELD);
function MAX is new UNARY_OPERATOR(O_MAX,FIELD);
function MIN is new UNARY_OPERATOR(O_MIN,FIELD);

function COUNT is new UNARY_OPERATOR(O_COUNT,FIELD);
function COUNT(X : STAR_TYPE) return FIELD;

function IS_IN is new BINARY_OPERATOR(O_IN,FIELD,   FIELD);
function IS_IN is new BINARY_OPERATOR(O_IN,INTEGER,FIELD);
function IS_IN is new BINARY_OPERATOR(O_IN,FLOAT,   FIELD);
function IS_IN is new BINARY_OPERATOR(O_IN,STRING,  FIELD);

function EXISTS is new UNARY_OPERATOR(O_EXISTS,FIELD);

function DESC is new UNARY_OPERATOR(O_DESC,FIELD);

function "and" is new BINARY_OPERATOR(O_AND,FIELD,FIELD);

function "and" is new BINARY_OPERATOR(O_AND,INTEGER,INTEGER);
function "and" is new BINARY_OPERATOR(O_AND,FLOAT,   FLOAT);
function "and" is new BINARY_OPERATOR(O_AND,STRING,  STRING);
function "and" is new BINARY_OPERATOR(O_AND,INTEGER,FLOAT);
function "and" is new BINARY_OPERATOR(O_AND,INTEGER,STRING);
function "and" is new BINARY_OPERATOR(O_AND,FLOAT,   INTEGER);
function "and" is new BINARY_OPERATOR(O_AND,FLOAT,   STRING);
function "and" is new BINARY_OPERATOR(O_AND,STRING,  INTEGER);
function "and" is new BINARY_OPERATOR(O_AND,STRING,  FLOAT);
function "and" is new BINARY_OPERATOR(O_AND,INTEGER,FIELD);
function "and" is new BINARY_OPERATOR(O_AND,FLOAT,   FIELD);
function "and" is new BINARY_OPERATOR(O_AND,STRING,  FIELD);
function "and" is new BINARY_OPERATOR(O_AND,FIELD,   INTEGER);
function "and" is new BINARY_OPERATOR(O_AND,FIELD,   FLOAT);
function "and" is new BINARY_OPERATOR(O_AND,FIELD,   STRING);

function "xor" is new BINARY_OPERATOR(O_XOR,FIELD,FIELD);

function "or" is new BINARY_OPERATOR(O_OR,FIELD,FIELD);

function "or" is new BINARY_OPERATOR(O_OR,INTEGER,INTEGER);
function "or" is new BINARY_OPERATOR(O_OR,FLOAT,   FLOAT);
function "or" is new BINARY_OPERATOR(O_OR,STRING,  STRING);
function "or" is new BINARY_OPERATOR(O_OR,INTEGER,FLOAT);
function "or" is new BINARY_OPERATOR(O_OR,FLOAT,   INTEGER);
function "or" is new BINARY_OPERATOR(O_OR,FIELD,   INTEGER);
function "or" is new BINARY_OPERATOR(O_OR,FIELD,   FLOAT);
function "or" is new BINARY_OPERATOR(O_OR,FIELD,   STRING);
function "or" is new BINARY_OPERATOR(O_OR,INTEGER,FIELD);
function "or" is new BINARY_OPERATOR(O_OR,FLOAT,   FIELD);
function "or" is new BINARY_OPERATOR(O_OR,STRING,  FIELD);

function EQ is new BINARY_OPERATOR(O_EQ,FIELD,   FIELD);
function EQ is new BINARY_OPERATOR(O_EQ,INTEGER,INTEGER);
function EQ is new BINARY_OPERATOR(O_EQ,FLOAT   ,FLOAT);
function EQ is new BINARY_OPERATOR(O_EQ,STRING,  STRING);
function EQ is new BINARY_OPERATOR(O_EQ,INTEGER,FLOAT);
function EQ is new BINARY_OPERATOR(O_EQ,FLOAT,   INTEGER);
```

```
function EQ is new BINARY_OPERATOR(O_EQ,FIELD,   INTEGER);
function EQ is new BINARY_OPERATOR(O_EQ,FIELD,   FLOAT);
function EQ is new BINARY_OPERATOR(O_EQ,FIELD,   STRING);
function EQ is new BINARY_OPERATOR(O_EQ,INTEGER,FIELD);
function EQ is new BINARY_OPERATOR(O_EQ,FLOAT,   FIELD);
function EQ is new BINARY_OPERATOR(O_EQ,STRING,  FIELD);

function NE is new BINARY_OPERATOR(O_NE,FIELD,   FIELD);
function NE is new BINARY_OPERATOR(O_NE,INTEGER,INTEGER);
function NE is new BINARY_OPERATOR(O_NE,FLOAT,   FLOAT);
function NE is new BINARY_OPERATOR(O_NE,STRING,  STRING);
function NE is new BINARY_OPERATOR(O_NE,INTEGER,FLOAT);
function NE is new BINARY_OPERATOR(O_NE,FLOAT,   INTEGER);
function NE is new BINARY_OPERATOR(O_NE,FIELD,   INTEGER);
function NE is new BINARY_OPERATOR(O_NE,FIELD,   FLOAT);
function NE is new BINARY_OPERATOR(O_NE,FIELD,   STRING);
function NE is new BINARY_OPERATOR(O_NE,INTEGER,FIELD);
function NE is new BINARY_OPERATOR(O_NE,FLOAT,   FIELD);
function NE is new BINARY_OPERATOR(O_NE,STRING,  FIELD);

function "<" is new BINARY_OPERATOR(O_LT,FIELD,   FIELD);
function "<" is new BINARY_OPERATOR(O_LT,INTEGER,INTEGER);
function "<" is new BINARY_OPERATOR(O_LT,FLOAT,   FLOAT);
function "<" is new BINARY_OPERATOR(O_LT,STRING,  STRING);
function "<" is new BINARY_OPERATOR(O_LT,INTEGER,FLOAT);
function "<" is new BINARY_OPERATOR(O_LT,FLOAT,   INTEGER);
function "<" is new BINARY_OPERATOR(O_LT,FIELD,   INTEGER);
function "<" is new BINARY_OPERATOR(O_LT,FIELD,   FLOAT);
function "<" is new BINARY_OPERATOR(O_LT,FIELD,   STRING);
function "<" is new BINARY_OPERATOR(O_LT,INTEGER,FIELD);
function "<" is new BINARY_OPERATOR(O_LT,FLOAT,   FIELD);
function "<" is new BINARY_OPERATOR(O_LT,STRING,  FIELD);

function "<=" is new BINARY_OPERATOR(O_LE,FIELD,   FIELD);
function "<=" is new BINARY_OPERATOR(O_LE,INTEGER,INTEGER);
function "<=" is new BINARY_OPERATOR(O_LE,FLOAT,   FLOAT);
function "<=" is new BINARY_OPERATOR(O_LE,STRING,  STRING);
function "<=" is new BINARY_OPERATOR(O_LE,INTEGER,FLOAT);
function "<=" is new BINARY_OPERATOR(O_LE,FLOAT,   INTEGER);
function "<=" is new BINARY_OPERATOR(O_LE,FIELD,   INTEGER);
function "<=" is new BINARY_OPERATOR(O_LE,FIELD,   FLOAT);
function "<=" is new BINARY_OPERATOR(O_LE,FIELD,   STRING);
function "<=" is new BINARY_OPERATOR(O_LE,INTEGER,FIELD);
function "<=" is new BINARY_OPERATOR(O_LE,FLOAT,   FIELD);
function "<=" is new BINARY_OPERATOR(O_LE,STRING,  FIELD);

function ">" is new BINARY_OPERATOR(O_GT,FIELD,   FIELD);
function ">" is new BINARY_OPERATOR(O_GT,INTEGER,INTEGER);
function ">" is new BINARY_OPERATOR(O_GT,FLOAT,   FLOAT);
function ">" is new BINARY_OPERATOR(O_GT,STRING,  STRING);
function ">" is new BINARY_OPERATOR(O_GT,INTEGER,FLOAT);
function ">" is new BINARY_OPERATOR(O_GT,FLOAT,   INTEGER);
function ">" is new BINARY_OPERATOR(O_GT,FIELD,   INTEGER);
function ">" is new BINARY_OPERATOR(O_GT,FIELD,   FLOAT);
function ">" is new BINARY_OPERATOR(O_GT,FIELD,   STRING);
function ">" is new BINARY_OPERATOR(O_GT,INTEGER,FIELD);
function ">" is new BINARY_OPERATOR(O_GT,FLOAT,   FIELD);
```

```
function ">" is new BINARY_OPERATOR(O_GT,STRING, FIELD);

function ">=" is new BINARY_OPERATOR(O_GE,FIELD,   FIELD);
function ">=" is new BINARY_OPERATOR(O_GE,INTEGER,INTEGER);
function ">=" is new BINARY_OPERATOR(O_GE,FLOAT,   FLOAT);
function ">=" is new BINARY_OPERATOR(O_GE,STRING,  STRING);
function ">=" is new BINARY_OPERATOR(O_GE,INTEGER,FLOAT);
function ">=" is new BINARY_OPERATOR(O_GE,FLOAT,   INTEGER);
function ">=" is new BINARY_OPERATOR(O_GE,FIELD,   INTEGER);
function ">=" is new BINARY_OPERATOR(O_GE,FIELD,   FLOAT);
function ">=" is new BINARY_OPERATOR(O_GE,FIELD,   STRING);
function ">=" is new BINARY_OPERATOR(O_GE,INTEGER,FIELD);
function ">=" is new BINARY_OPERATOR(O_GE,FLOAT,   FIELD);
function ">=" is new BINARY_OPERATOR(O_GE,STRING,  FIELD);

function "+" is new BINARY_OPERATOR(O_PLUS,FIELD,   FIELD);
function "+" is new BINARY_OPERATOR(O_PLUS,INTEGER,INTEGER);
function "+" is new BINARY_OPERATOR(O_PLUS,FLOAT,   FLOAT);
function "+" is new BINARY_OPERATOR(O_PLUS,INTEGER,FLOAT);
function "+" is new BINARY_OPERATOR(O_PLUS,FLOAT,   INTEGER);
function "+" is new BINARY_OPERATOR(O_PLUS,FIELD,   INTEGER);
function "+" is new BINARY_OPERATOR(O_PLUS,FIELD,   FLOAT);
function "+" is new BINARY_OPERATOR(O_PLUS,INTEGER,FIELD);
function "+" is new BINARY_OPERATOR(O_PLUS,FLOAT,   FIELD);

function "-" is new BINARY_OPERATOR(O_MINUS,FIELD,   FIELD);
function "-" is new BINARY_OPERATOR(O_MINUS,INTEGER,INTEGER);
function "-" is new BINARY_OPERATOR(O_MINUS,FLOAT,   FLOAT);
function "-" is new BINARY_OPERATOR(O_MINUS,INTEGER,FLOAT);
function "-" is new BINARY_OPERATOR(O_MINUS,FLOAT,   INTEGER);
function "-" is new BINARY_OPERATOR(O_MINUS,FIELD,   INTEGER);
function "-" is new BINARY_OPERATOR(O_MINUS,FIELD,   FLOAT);
function "-" is new BINARY_OPERATOR(O_MINUS,INTEGER,FIELD);
function "-" is new BINARY_OPERATOR(O_MINUS,FLOAT,   FIELD);

function "&"(L : TABLE; R : TABLE) return TABLE renames SQL_DEFINITIONS."&";

function "&" is new BINARY_OPERATOR(O_CAT,FIELD,   FIELD);
function "&" is new BINARY_OPERATOR(O_CAT,INTEGER,INTEGER);
function "&" is new BINARY_OPERATOR(O_CAT,FLOAT,   FLOAT);
function "&" is new BINARY_OPERATOR(O_CAT,STRING,  STRING);
function "&" is new BINARY_OPERATOR(O_CAT,INTEGER,FLOAT);
function "&" is new BINARY_OPERATOR(O_CAT,INTEGER,STRING);
function "&" is new BINARY_OPERATOR(O_CAT,FLOAT,   INTEGER);
function "&" is new BINARY_OPERATOR(O_CAT,FLOAT,   STRING);
function "&" is new BINARY_OPERATOR(O_CAT,STRING,  INTEGER);
function "&" is new BINARY_OPERATOR(O_CAT,STRING,  FLOAT);
function "&" is new BINARY_OPERATOR(O_CAT,INTEGER,FIELD);
function "&" is new BINARY_OPERATOR(O_CAT,FLOAT,   FIELD);
function "&" is new BINARY_OPERATOR(O_CAT,STRING,  FIELD);
function "&" is new BINARY_OPERATOR(O_CAT,FIELD,   INTEGER);
function "&" is new BINARY_OPERATOR(O_CAT,FIELD,   FLOAT);
function "&" is new BINARY_OPERATOR(O_CAT,FIELD,   STRING);

function "+" is new UNARY_OPERATOR(O_UNARY_PLUS,FIELD);
function "+" is new UNARY_OPERATOR(O_UNARY_PLUS,INTEGER);
function "+" is new UNARY_OPERATOR(O_UNARY_PLUS,FLOAT);
```

```
function "-" is new UNARY_OPERATOR(O_UNARY_MINUS,FIELD);
function "-" is new UNARY_OPERATOR(O_UNARY_MINUS,INTEGER);
function "-" is new UNARY_OPERATOR(O_UNARY_MINUS,FLOAT);

function "*" is new BINARY_OPERATOR(O_TIMES,FIELD,   FIELD);
function "*" is new BINARY_OPERATOR(O_TIMES,INTEGER,INTEGER);
function "*" is new BINARY_OPERATOR(O_TIMES,FLOAT,   FLOAT);
function "*" is new BINARY_OPERATOR(O_TIMES,INTEGER,FLOAT);
function "*" is new BINARY_OPERATOR(O_TIMES,FLOAT,   INTEGER);
function "*" is new BINARY_OPERATOR(O_TIMES,FIELD,   INTEGER);
function "*" is new BINARY_OPERATOR(O_TIMES,FIELD,   FLOAT);
function "*" is new BINARY_OPERATOR(O_TIMES,INTEGER,FIELD);
function "*" is new BINARY_OPERATOR(O_TIMES,FLOAT,   FIELD);

function "/" is new BINARY_OPERATOR(O_DIV,FIELD,   FIELD);
function "/" is new BINARY_OPERATOR(O_DIV,INTEGER,INTEGER);
function "/" is new BINARY_OPERATOR(O_DIV,FLOAT,   FLOAT);
function "/" is new BINARY_OPERATOR(O_DIV,INTEGER,FLOAT);
function "/" is new BINARY_OPERATOR(O_DIV,FLOAT,   INTEGER);
function "/" is new BINARY_OPERATOR(O_DIV,FIELD,   INTEGER);
function "/" is new BINARY_OPERATOR(O_DIV,FIELD,   FLOAT);
function "/" is new BINARY_OPERATOR(O_DIV,INTEGER,FIELD);
function "/" is new BINARY_OPERATOR(O_DIV,FLOAT,   FIELD);

function "mod" is new BINARY_OPERATOR(O_MOD,FIELD,   FIELD);
function "mod" is new BINARY_OPERATOR(O_MOD,INTEGER,INTEGER);
function "mod" is new BINARY_OPERATOR(O_MOD,FIELD,   INTEGER);
function "mod" is new BINARY_OPERATOR(O_MOD,INTEGER,FIELD);

function "rem" is new BINARY_OPERATOR(O_REM,FIELD,   FIELD);
function "rem" is new BINARY_OPERATOR(O_REM,INTEGER,INTEGER);
function "rem" is new BINARY_OPERATOR(O_REM,FIELD,   INTEGER);
function "rem" is new BINARY_OPERATOR(O_REM,INTEGER,FIELD);

function "**" is new BINARY_OPERATOR(O_POWER,FIELD,   FIELD);
function "**" is new BINARY_OPERATOR(O_POWER,INTEGER,INTEGER);
function "**" is new BINARY_OPERATOR(O_POWER,FLOAT,   INTEGER);
function "**" is new BINARY_OPERATOR(O_POWER,FIELD,   INTEGER);
function "**" is new BINARY_OPERATOR(O_POWER,INTEGER,FIELD);
function "**" is new BINARY_OPERATOR(O_POWER,FLOAT,   FIELD);

function "abs" is new UNARY_OPERATOR(O_ABS,FIELD);
function "abs" is new UNARY_OPERATOR(O_ABS,INTEGER);
function "abs" is new UNARY_OPERATOR(O_ABS,FLOAT);

function "not" is new UNARY_OPERATOR(O_NOT,FIELD);

subtype DATABASE_TYPE is SQL_DEFINITIONS.DATABASE_TYPE;
subtype CURSOR_TYPE   is SQL_DEFINITIONS.CURSOR_TYPE;
subtype FIELD_INDEX   is SQL_DEFINITIONS.FIELD_INDEX;

function EXECUTE(F : FIELD) return CURSOR_TYPE
    renames SQL_DEFINITIONS.EXECUTE;

procedure EXECUTE(F : in FIELD) renames SQL_DEFINITIONS.EXECUTE;
procedure LIST   (F : in FIELD) renames SQL_DEFINITIONS.LIST;
procedure SHOW   (F : in FIELD) renames SQL_DEFINITIONS.SHOW;
```

```
  procedure SET_DATABASE(DB : in DATABASE_TYPE)
      renames SQL_DEFINITIONS.SET_DATABASE;

  procedure NEXT_RECORD(CURSOR : in out CURSOR_TYPE)
      renames SQL_DEFINITIONS.NEXT_RECORD;

  procedure FETCH(CURSOR : in  CURSOR_TYPE;
                  FIELD  : in  FIELD_INDEX;
                  INT    : out INTEGER) renames SQL_DEFINITIONS.FETCH;

  procedure FETCH(CURSOR : in  CURSOR_TYPE;
                  FIELD  : in  FIELD_INDEX;
                  FLT    : out FLOAT) renames SQL_DEFINITIONS.FETCH;

  procedure FETCH(CURSOR : in  CURSOR_TYPE;
                  FIELD  : in  FIELD_INDEX;
                  STR    : out STRING;
                  LAST   : out NATURAL) renames SQL_DEFINITIONS.FETCH;

  function FETCH(CURSOR : CURSOR_TYPE;
                 FIELD  : FIELD_INDEX) return INTEGER
      renames SQL_DEFINITIONS.FETCH;

  function FETCH(CURSOR : CURSOR_TYPE;
                 FIELD  : FIELD_INDEX) return FLOAT
      renames SQL_DEFINITIONS.FETCH;

  function FETCH(CURSOR : CURSOR_TYPE;
                 FIELD  : FIELD_INDEX) return STRING
      renames SQL_DEFINITIONS.FETCH;

  function LOAD_DATABASE(FILE_NAME : in STRING) return DATABASE_TYPE
      renames SQL_DEFINITIONS.LOAD_DATABASE;

  procedure SAVE_DATABASE(FILE_NAME : in STRING;
                          DATABASE  : in DATABASE_TYPE)
      renames SQL_DEFINITIONS.SAVE_DATABASE;

  CALL_ERROR          : exception renames SQL_DEFINITIONS.CALL_ERROR;
  DONE_ERROR          : exception renames SQL_DEFINITIONS.DONE_ERROR;
  FIELD_ERROR         : exception renames SQL_DEFINITIONS.FIELD_ERROR;
  SYNTAX_ERROR        : exception renames SQL_DEFINITIONS.SYNTAX_ERROR;
  TABLE_ERROR         : exception renames SQL_DEFINITIONS.TABLE_ERROR;
  TRUNCATE_ERROR      : exception renames SQL_DEFINITIONS.TRUNCATE_ERROR;
  TYPE_ERROR          : exception renames SQL_DEFINITIONS.TYPE_ERROR;
  UNIMPLEMENTED_ERROR : exception renames SQL_DEFINITIONS.UNIMPLEMENTED_ERROR;

end SQL_OPERATIONS;

with SQL_DEFINITIONS;
  use SQL_DEFINITIONS;

package body SQL_OPERATIONS is

  function SELEC(WHAT  : STAR_TYPE;
                 FROM  : TABLE  := NULL_TABLE;
                 WHERE : FIELD  := NULL_FIELD;
```

```
                    GROUP  : FIELD := NULL_FIELD;
                    HAVING : FIELD := NULL_FIELD;
                    ORDER  : FIELD := NULL_FIELD) return FIELD is
  begin
    return SELEC(STAR,FROM,WHERE,GROUP,HAVING,ORDER);
  end SELEC;

  function COUNT(X : STAR_TYPE) return FIELD is
  begin
    return COUNT(STAR);
  end COUNT;

end SQL_OPERATIONS;
```

```
with SQL_DEFINITIONS;
  use SQL_DEFINITIONS;

package DATE_UNDERLYING is

  type CELLAR_TYPE is
    record
      STAR, BIN, WINE, PRODUCER, YEAR, BOTTLES, READY, COMMENTS : FIELD;
    end record;

  type FLIGHTS_TYPE is
    record
      STAR, FLIGHT, FROM_CODE, TO_CODE, DEP_TIME, ARR_TIME : FIELD;
    end record;

  type CITIES_TYPE is
    record
      STAR, CODE, CITY : FIELD;
    end record;

  type PARCELS_TYPE is
    record
      STAR, APN, ROAD, OWNER, IMPROVED, LAST_ENTRY, BALANCE : FIELD;
    end record;

  type OWNERS_TYPE is
    record
      STAR, OWNER, ADDRESS, PHONE : FIELD;
    end record;

  type PARCEL_ACCOUNTS_TYPE is
    record
      STAR, APN, EN_TRY, DATE, DESCRIPTION, TYP, AMOUNT, BALANCE : FIELD;
    end record;

  type SPECIAL_ASSESSMENTS_TYPE is
    record
      STAR, SAN, ROAD, DATE, TOTAL, PER_PARCEL, EXPLANATION, PAYEE : FIELD;
    end record;

  type LEDGER_TYPE is
    record
      STAR, EN_TRY, DATE, DESCRIPTION, TYP, PARTY, AMOUNT, BALANCE : FIELD;
    end record;

  type GENERAL_LEDGER_TYPE is new LEDGER_TYPE;
  type REDWOOD_LEDGER_TYPE is new LEDGER_TYPE;
  type CREEK_LEDGER_TYPE   is new LEDGER_TYPE;
  type MILL_LEDGER_TYPE    is new LEDGER_TYPE;

  type LAST_ENTRIES_TYPE is
    record
      STAR, ACCOUNT, EN_TRY, BALANCE : FIELD;
    end record;

  type CELLAR_TABLE              is access CELLAR_TYPE;
  type FLIGHTS_TABLE             is access FLIGHTS_TYPE;
```

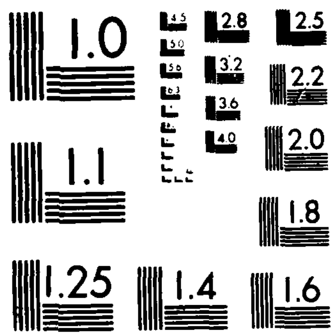
```
type CITIES_TABLE              is access CITIES_TYPE;
type PARCELS_TABLE             is access PARCELS_TYPE;
type OWNERS_TABLE              is access OWNERS_TYPE;
type PARCEL_ACCOUNTS_TABLE     is access PARCEL_ACCOUNTS_TYPE;
type SPECIAL_ASSESSMENTS_TABLE is access SPECIAL_ASSESSMENTS_TYPE;
type GENERAL_LEDGER_TABLE      is access GENERAL_LEDGER_TYPE;
type REDWOOD_LEDGER_TABLE      is access REDWOOD_LEDGER_TYPE;
type CREEK_LEDGER_TABLE        is access CREEK_LEDGER_TYPE;
type MILL_LEDGER_TABLE         is access MILL_LEDGER_TYPE;
type LAST_ENTRIES_TABLE        is access LAST_ENTRIES_TYPE;

BIN         : constant FIELD := MAKE_FIELD("BIN");
WINE        : constant FIELD := MAKE_FIELD("WINE");
PRODUCER    : constant FIELD := MAKE_FIELD("PRODUCER");
YEAR        : constant FIELD := MAKE_FIELD("YEAR");
BOTTLES     : constant FIELD := MAKE_FIELD("BOTTLES");
READY       : constant FIELD := MAKE_FIELD("READY");
COMMENTS    : constant FIELD := MAKE_FIELD("COMMENTS");
FLIGHT      : constant FIELD := MAKE_FIELD("FLIGHT");
FROM_CODE   : constant FIELD := MAKE_FIELD("FROM_CODE");
TO_CODE     : constant FIELD := MAKE_FIELD("TO_CODE");
DEP_TIME    : constant FIELD := MAKE_FIELD("DEP_TIME");
ARR_TIME    : constant FIELD := MAKE_FIELD("ARR_TIME");
CODE        : constant FIELD := MAKE_FIELD("CODE");
CITY        : constant FIELD := MAKE_FIELD("CITY");
APN         : constant FIELD := MAKE_FIELD("APN");
ROAD        : constant FIELD := MAKE_FIELD("ROAD");
OWNER       : constant FIELD := MAKE_FIELD("OWNER");
IMPROVED    : constant FIELD := MAKE_FIELD("IMPROVED");
LAST_ENTRY  : constant FIELD := MAKE_FIELD("LAST_ENTRY");
BALANCE     : constant FIELD := MAKE_FIELD("BALANCE");
ADDRESS     : constant FIELD := MAKE_FIELD("ADDRESS");
PHONE       : constant FIELD := MAKE_FIELD("PHONE");
EN_TRY      : constant FIELD := MAKE_FIELD("ENTRY");
DATE        : constant FIELD := MAKE_FIELD("DATE");
DESCRIPTION : constant FIELD := MAKE_FIELD("DESCRIPTION");
TYP         : constant FIELD := MAKE_FIELD("TYPE");
AMOUNT      : constant FIELD := MAKE_FIELD("AMOUNT");
SAN         : constant FIELD := MAKE_FIELD("SAN");
TOTAL       : constant FIELD := MAKE_FIELD("TOTAL");
PER_PARCEL  : constant FIELD := MAKE_FIELD("PER_PARCEL");
EXPLANATION : constant FIELD := MAKE_FIELD("EXPLANATION");
PAYEE       : constant FIELD := MAKE_FIELD("PAYEE");
PARTY       : constant FIELD := MAKE_FIELD("PARTY");
ACCOUNT     : constant FIELD := MAKE_FIELD("ACCOUNT");

CELLAR_DATA               : CELLAR_TABLE;
FLIGHTS_DATA              : FLIGHTS_TABLE;
CITIES_DATA               : CITIES_TABLE;
PARCELS_DATA              : PARCELS_TABLE;
OWNERS_DATA               : OWNERS_TABLE;
PARCEL_ACCOUNTS_DATA      : PARCEL_ACCOUNTS_TABLE;
SPECIAL_ASSESSMENTS_DATA  : SPECIAL_ASSESSMENTS_TABLE;
GENERAL_LEDGER_DATA       : GENERAL_LEDGER_TABLE;
REDWOOD_LEDGER_DATA       : REDWOOD_LEDGER_TABLE;
CREEK_LEDGER_DATA         : CREEK_LEDGER_TABLE;
MILL_LEDGER_DATA          : MILL_LEDGER_TABLE;
```

```
  LAST_ENTRIES_DATA          : LAST_ENTRIES_TABLE;

  procedure CELLAR              (X : in out CELLAR_TABLE);
  procedure FLIGHTS             (X : in out FLIGHTS_TABLE);
  procedure CITIES              (X : in out CITIES_TABLE);
  procedure PARCELS             (X : in out PARCELS_TABLE);
  procedure OWNERS              (X : in out OWNERS_TABLE);
  procedure PARCEL_ACCOUNTS     (X : in out PARCEL_ACCOUNTS_TABLE);
  procedure SPECIAL_ASSESSMENTS (X : in out SPECIAL_ASSESSMENTS_TABLE);
  procedure GENERAL_LEDGER      (X : in out GENERAL_LEDGER_TABLE);
  procedure REDWOOD_LEDGER      (X : in out REDWOOD_LEDGER_TABLE);
  procedure CREEK_LEDGER        (X : in out CREEK_LEDGER_TABLE);
  procedure MILL_LEDGER         (X : in out MILL_LEDGER_TABLE);
  procedure LAST_ENTRIES        (X : in out LAST_ENTRIES_TABLE);

end DATE_UNDERLYING;

with SQL_DEFINITIONS;
  use SQL_DEFINITIONS;

package body DATE_UNDERLYING is

  procedure CELLAR(X : in out CELLAR_TABLE) is
    T : TABLE_NAME;
  begin
    if X = null then
      T := MAKE_TABLE_NAME("CELLAR");
      X := new CELLAR_TYPE'(
        MAKE_FIELD(T.STAR),
        MAKE_FIELD(T.BIN),
        MAKE_FIELD(T.WINE),
        MAKE_FIELD(T.PRODUCER),
        MAKE_FIELD(T.YEAR),
        MAKE_FIELD(T.BOTTLES),
        MAKE_FIELD(T.READY),
        MAKE_FIELD(T.COMMENTS) );
    end if;
  end CELLAR;

  procedure FLIGHTS(X : in out FLIGHTS_TABLE) is
    T : TABLE_NAME;
  begin
    if X = null then
      T := MAKE_TABLE_NAME("FLIGHTS");
      X := new FLIGHTS_TYPE'(
        MAKE_FIELD(T.STAR),
        MAKE_FIELD(T.FLIGHT),
        MAKE_FIELD(T.FROM_CODE),
        MAKE_FIELD(T.TO_CODE),
        MAKE_FIELD(T.DEP_TIME),
        MAKE_FIELD(T.ARR_TIME) );
    end if;
  end FLIGHTS;

  procedure CITIES(X : in out CITIES_TABLE) is
    T : TABLE_NAME;
  begin
```

```
    if X = null then
      T := MAKE_TABLE_NAME("CITIES");
      X := new CITIES_TYPE'(
        MAKE_FIELD(T,STAR),
        MAKE_FIELD(T,CODE),
        MAKE_FIELD(T,CITY) );
    end if;
  end CITIES;

  procedure PARCELS(X : in out PARCELS_TABLE) is
    T : TABLE_NAME;
  begin
    if X = null then
      T := MAKE_TABLE_NAME("PARCELS");
      X := new PARCELS_TYPE'(
        MAKE_FIELD(T,STAR),
        MAKE_FIELD(T,APN),
        MAKE_FIELD(T,ROAD),
        MAKE_FIELD(T,OWNER),
        MAKE_FIELD(T,IMPROVED),
        MAKE_FIELD(T,LAST_ENTRY),
        MAKE_FIELD(T,BALANCE) );
    end if;
  end PARCELS;

  procedure OWNERS(X : in out OWNERS_TABLE) is
    T : TABLE_NAME;
  begin
    if X = null then
      T := MAKE_TABLE_NAME("OWNERS");
      X := new OWNERS_TYPE'(
        MAKE_FIELD(T,STAR),
        MAKE_FIELD(T,OWNER),
        MAKE_FIELD(T,ADDRESS),
        MAKE_FIELD(T,PHONE) );
    end if;
  end OWNERS;

  procedure PARCEL_ACCOUNTS(X : in out PARCEL_ACCOUNTS_TABLE) is
    T : TABLE_NAME;
  begin
    if X = null then
      T := MAKE_TABLE_NAME("PARCEL_ACCOUNTS");
      X := new PARCEL_ACCOUNTS_TYPE'(
        MAKE_FIELD(T,STAR),
        MAKE_FIELD(T,APN),
        MAKE_FIELD(T,EN_TRY),
        MAKE_FIELD(T,DATE),
        MAKE_FIELD(T,DESCRIPTION),
        MAKE_FIELD(T,TYP),
        MAKE_FIELD(T,AMOUNT),
        MAKE_FIELD(T,BALANCE) );
    end if;
  end PARCEL_ACCOUNTS;

  procedure SPECIAL_ASSESSMENTS(X : in out SPECIAL A[illegible]
    T : TABLE_NAME
```

AD-A182 885 PRELIMINARY VERSION: ADA (TRADE NAME)/SQL: A STANDARD PORTABLE ADA-DBMS I (U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA B R BRYKCZYNSKI ET AL APR 87 IDA-P-1944 3/3

UNCLASSIFIED IDA/HQ-86-31239 MDA903-84-C-0031 F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

```
begin
  if X = null then
    T := MAKE_TABLE_NAME("SPECIAL_ASSESSMENTS");
    X := new SPECIAL_ASSESSMENTS_TYPE'(
      MAKE_FIELD(T,STAR),
      MAKE_FIELD(T,SAN),
      MAKE_FIELD(T,ROAD),
      MAKE_FIELD(T,DATE),
      MAKE_FIELD(T,TOTAL),
      MAKE_FIELD(T,PER_PARCEL),
      MAKE_FIELD(T,EXPLANATION),
      MAKE_FIELD(T,PAYEE) );
  end if;
end SPECIAL_ASSESSMENTS;

procedure GENERAL_LEDGER(X : in out GENERAL_LEDGER_TABLE) is
  T : TABLE_NAME;
begin
  if X = null then
    T := MAKE_TABLE_NAME("GENERAL_LEDGER");
    X := new GENERAL_LEDGER_TYPE'(
      MAKE_FIELD(T,STAR),
      MAKE_FIELD(T,EN_TRY),
      MAKE_FIELD(T,DATE),
      MAKE_FIELD(T,DESCRIPTION),
      MAKE_FIELD(T,TYP),
      MAKE_FIELD(T,PARTY),
      MAKE_FIELD(T,AMOUNT),
      MAKE_FIELD(T,BALANCE) );
  end if;
end GENERAL_LEDGER;

procedure REDWOOD_LEDGER(X : in out REDWOOD_LEDGER_TABLE) is
  T : TABLE_NAME;
begin
  if X = null then
    T := MAKE_TABLE_NAME("REDWOOD_LEDGER");
    X := new RED_FIELD(T,EN_TRY),
      MAKE_FIELD(T,DATE),
      MAKE_FIELD(T,DESCRIPTION),
      MAKE_FIELD(T,TYP),
      MAKE_FIELD(T,PARTY),
      MAKE_FIELD(T,AMOUNT),
      MAKE_FIELD(T,BALANCE) );
  end if;
end REDWOOD_LEDGER;

procedure CREEK_LEDGER(X : in out CREEK_LEDGER_TABLE) is
  T : TABLE_NAME;
begin
  if X = null then
    T := MAKE_TABLE_NAME("CREEK_LEDGER");
    X := new CREEK_LEDGER_TYPE'(
      MAKE_FIELD(T,STAR),
      MAKE_FIELD(T,EN_TRY),
      MAKE_FIELD(T,DATE),
      MAKE_FIELD(T,DESCRIPTION),
```

```
          MAKE_FIELD(T,TYP),
          MAKE_FIELD(T,PARTY),
          MAKE_FIELD(T,AMOUNT),
          MAKE_FIELD(T,BALANCE) );
      end if;
    end CREEK_LEDGER;

    procedure MILL_LEDGER(X : in out MILL_LEDGER_TABLE) is
      T : TABLE_NAME;
    begin
      if X = null then
        T := MAKE_TABLE_NAME("MILL_LEDGER");
        X := new MILL_LEDGER_TYPE'(
          MAKE_FIELD(T,STAR),
          MAKE_FIELD(T,EN_TRY),
          MAKE_FIELD(T,DATE),
          MAKE_FIELD(T,DESCRIPTION),
          MAKE_FIELD(T,TYP),
          MAKE_FIELD(T,PARTY),
          MAKE_FIELD(T,AMOUNT),
          MAKE_FIELD(T,BALANCE) );
      end if;
    end MILL_LEDGER;

    procedure LAST_ENTRIES(X : in out LAST_ENTRIES_TABLE) is
      T : TABLE_NAME;
    begin
      if X = null then
        T := MAKE_TABLE_NAME("LAST_ENTRIES");
        X := new LAST_ENTRIES_TYPE'(
          MAKE_FIELD(T,STAR),
          MAKE_FIELD(T,ACCOUNT),
          MAKE_FIELD(T,EN_TRY),
          MAKE_FIELD(T,BALANCE) );
      end if;
    end LAST_ENTRIES;

  begin

    CELLAR              (CELLAR_DATA);
    FLIGHTS             (FLIGHTS_DATA);
    CITIES              (CITIES_DATA);
    PARCELS             (PARCELS_DATA);
    OWNERS              (OWNERS_DATA);
    PARCEL_ACCOUNTS     (PARCEL_ACCOUNTS_DATA);
    SPECIAL_ASSESSMENTS (SPECIAL_ASSESSMENTS_DATA);
    GENERAL_LEDGER      (GENERAL_LEDGER_DATA);
    REDWOOD_LEDGER      (REDWOOD_LEDGER_DATA);
    CREEK_LEDGER        (CREEK_LEDGER_DATA);
    MILL_LEDGER         (MILL_LEDGER_DATA);
    LAST_ENTRIES        (LAST_ENTRIES_DATA);

  end DATE_UNDERLYING;
```

```
separate(SQL_DEFINITIONS.SQL_FUNCTIONS)
package body PROGRAM_FUNCTIONS is

  DATABASE : DATABASE_TYPE;

  MATCHING_TYPES : constant array(DATABASE_FIELD_TYPE) of FIELD_TYPE_TYPE :=
    ( INTEGER_FIELD => INTEGER_LITERAL,
      FLOAT_FIELD   => FLOAT_LITERAL,
      STRING_FIELD  => STRING_LITERAL);

  function EQUAL(LEFT, RIGHT : VALUE_LINK) return BOOLEAN is
  begin
    case LEFT.FIELD_TYPE is
      when STRING_FIELD =>
        return LEFT.STRING_VALUE.all = RIGHT.STRING_VALUE.all;
      when others =>
        return LEFT.all = RIGHT.all;
    end case;
  exception
    when CONSTRAINT_ERROR =>
      return FALSE;
  end EQUAL;

  function FIND_TABLE(TABLE : TABLE_NAME)
      return TABLE_LINK is
  begin
    for I in 1..DATABASE'LAST loop
      if TABLE.all = DATABASE(I).NAME.all then
        return DATABASE(I);
      end if;
    end loop;
    raise TABLE_ERROR;
  end FIND_TABLE;

  function FIND_FIELD(TABLE : TABLE_LINK; FIELD : FIELD_NAME)
      return FIELD_INDEX is
  begin
    for I in 1..TABLE.FIELDS'LAST loop
      if FIELD.all = TABLE.FIELDS(I).NAME.all then
        return I;
      end if;
    end loop;
    raise FIELD_ERROR;
  end FIND_FIELD;

  function CREATE_LITERAL_VALUE(VALUE : FIELD) return VALUE_LINK is
  begin
    case VALUE.FIELD_TYPE is
      when INTEGER_LITERAL =>
        return new VALUE_TYPE'(INTEGER_FIELD,VALUE.INTEGER_VALUE);
      when FLOAT_LITERAL =>
        return new VALUE_TYPE'(FLOAT_FIELD,VALUE.FLOAT_VALUE);
      when STRING_LITERAL =>
        return new VALUE_TYPE'(STRING_FIELD,VALUE.STRING_VALUE);
      when others =>
        raise UNIMPLEMENTED_ERROR;
    end case;
```

```
end CREATE_LITERAL_VALUE;

procedure BUILD_WHERE(CURSOR    : in out CURSOR_TYPE;
                      WHERE     : in FIELD;
                      FROM      : in TABLE_LINK) is
  FIELD_NUMBER : FIELD_INDEX;
  TARGET_TYPE  : DATABASE_FIELD_TYPE;
  LEFT,
  RIGHT        : FIELD;
begin
  case WHERE.FIELD_TYPE is
    when EMPTY =>
      return;
    when OPERATOR =>
      null;
    when others =>
      raise SYNTAX_ERROR;
  end case;
  LEFT  := WHERE.DOWN_LINK;
  RIGHT := LEFT.ACROSS_LINK;
  case WHERE.OPCODE is
    when O_AND =>
      BUILD_WHERE(CURSOR, RIGHT, FROM);
      BUILD_WHERE(CURSOR, LEFT, FROM);
    when O_EQ =>
      case LEFT.FIELD_TYPE is
        when QUALIFIED_FIELD =>
          if LEFT.RELATION.all /= FROM.NAME.all then
            raise FIELD_ERROR;
          end if;
        when UNQUALIFIED_FIELD =>
          null;
        when others =>
          raise UNIMPLEMENTED_ERROR;
      end case;
      FIELD_NUMBER := FIND_FIELD(FROM, LEFT.NAME);
      TARGET_TYPE  := FROM.FIELDS(FIELD_NUMBER).DATA_TYPE;
      if RIGHT.FIELD_TYPE /=
          MATCHING_TYPES(TARGET_TYPE) then
        raise UNIMPLEMENTED_ERROR;
      end if;
      CURSOR.QUERY := new QUERY_NODE_REC'(CURSOR.QUERY, FIELD_NUMBER,
          CREATE_LITERAL_VALUE(RIGHT));
    when others =>
      raise UNIMPLEMENTED_ERROR;
  end case;
end BUILD_WHERE;

function EXECUTE(F : FIELD) return CURSOR_TYPE is
  WHAT,
  FROM_FIELD,
  WHERE,
  CLAUSE     : FIELD;
  FROM       : TABLE;
  TABLE_PTR  : TABLE_LINK;
  CURSOR     : CURSOR_TYPE;
begin
```

```
    WHAT := F.DOWN_LINK;
    FROM_FIELD := WHAT.ACROSS_LINK;
    FROM := FROM_FIELD.TABLE_LINK;
    WHERE := FROM_FIELD.ACROSS_LINK;
    CLAUSE := WHERE;
    if DATABASE = null or else F.OPCODE /= O_SELECT then
      raise CALL_ERROR;
    elsif FROM.NEXT_LINK /= null or else WHAT.NAME.all /= "*" then
      raise UNIMPLEMENTED_ERROR;
    elsif WHAT.RELATION /= null and then WHAT.RELATION.all /= FROM.NAME.all
        then
      raise FIELD_ERROR;
    end if;
    for I in 1..3 loop
      CLAUSE := CLAUSE.ACROSS_LINK;
      if CLAUSE.FIELD_TYPE /= EMPTY then
        raise UNIMPLEMENTED_ERROR;
      end if;
    end loop;
    TABLE_PTR := FIND_TABLE(FROM.NAME);
    CURSOR.CURRENT_RECORD := TABLE_PTR.RECORDS;
    BUILD_WHERE(CURSOR,WHERE TABLE_PTR);
    return CURSOR;
  exception
    when CONSTRAINT_ERROR =>
      raise UNIMPLEMENTED_ERROR;
  end EXECUTE;

  procedure SET_DATABASE(DB : in DATABASE_TYPE) is
  begin
    DATABASE := DB;
  end SET_DATABASE;

  function EQUAL_RECORD(CURSOR : in CURSOR_TYPE) return BOOLEAN is
    COMPARE : QUERY_NODE := CURSOR.QUERY;
  begin
    while COMPARE /= null loop
      if not EQUAL(COMPARE.VALUE,CURSOR.CURRENT_RECORD.VALUES(COMPARE.FIELD))
          then
        return FALSE;
      end if;
      COMPARE := COMPARE.NEXT_NODE;
    end loop;
    return TRUE;
  end EQUAL_RECORD;

  procedure NEXT_RECORD(CURSOR : in out CURSOR_TYPE) is
  begin
    if CURSOR.CURRENT_RECORD = null then
      raise DONE_ERROR;
    elsif CURSOR.NEW_QUERY = TRUE then
      CURSOR.NEW_QUERY := FALSE;
    else
      CURSOR.CURRENT_RECORD := CURSOR.CURRENT_RECORD.NEXT_RECORD;
    end if;
    while CURSOR.CURRENT_RECORD /= null loop
      if EQUAL_RECORD(CURSOR) then
```

```
        return;
      end if;
      CURSOR.CURRENT_RECORD := CURSOR.CURRENT_RECORD.NEXT_RECORD;
    end loop;
    raise DONE_ERROR;
  end NEXT_RECORD;

  procedure FETCH_RAZOR(CURSOR : in CURSOR_TYPE; FIELD : in FIELD_INDEX) is
  begin
    if CURSOR.CURRENT_RECORD = null then
      raise CALL_ERROR;
    elsif FIELD > CURSOR.CURRENT_RECORD.VALUES'LAST then
      raise FIELD_ERROR;
    end if;
  end FETCH_RAZOR;

  function FETCH(CURSOR : in  CURSOR_TYPE;
                 FIELD  : in  FIELD_INDEX) return INTEGER is
  begin
    FETCH_RAZOR(CURSOR,FIELD);
    return CURSOR.CURRENT_RECORD.VALUES(FIELD).INTEGER_VALUE;
  exception
    when CONSTRAINT_ERROR =>
      raise TYPE_ERROR;
  end FETCH;

  function FETCH(CURSOR : in  CURSOR_TYPE;
                 FIELD  : in  FIELD_INDEX) return FLOAT is
  begin
    FETCH_RAZOR(CURSOR,FIELD);
    return CURSOR.CURRENT_RECORD.VALUES(FIELD).FLOAT_VALUE;
  exception
    when CONSTRAINT_ERROR =>
      raise TYPE_ERROR;
  end FETCH;

  function FETCH(CURSOR : in  CURSOR_TYPE;
                 FIELD  : in  FIELD_INDEX) return STRING is
  begin
    FETCH_RAZOR(CURSOR,FIELD);
    return CURSOR.CURRENT_RECORD.VALUES(FIELD).STRING_VALUE.all;
  exception
    when CONSTRAINT_ERROR =>
      raise TYPE_ERROR;
  end FETCH;

  procedure FETCH(CURSOR : in  CURSOR_TYPE;
                  FIELD  : in  FIELD_INDEX;
                  INT    : out INTEGER) is
  begin
    INT := FETCH(CURSOR,FIELD);
  end FETCH;

  procedure FETCH(CURSOR : in  CURSOR_TYPE;
                  FIELD  : in  FIELD_INDEX;
                  FLT    : out FLOAT) is
  begin
```

```
    FLT := FETCH(CURSOR,FIELD);
  end FETCH;

  procedure FETCH(CURSOR : in  CURSOR_TYPE;
                  FIELD  : in  FIELD_INDEX;
                  STR    : out STRING;
                  LAST   : out NATURAL) is
    S : STRING_LINK;
    L : NATURAL;
  begin
    FETCH_RAZOR(CURSOR,FIELD);
    S := CURSOR.CURRENT_RECORD.VALUES(FIELD).STRING_VALUE;
    if S'LENGTH > STR'LENGTH then
      raise TRUNCATE_ERROR;
    end if;
    L := STR'FIRST + S'LENGTH - 1;
    STR(STR'FIRST..L) := S.all;
    LAST := L;
  exception
    when CONSTRAINT_ERROR =>
      raise TYPE_ERROR;
  end FETCH;

  procedure MAKE_NEW_RECORD(TABLE : in out TABLE_LINK;
                            REC   : out RECORD_LINK) is
    NEW_RECORD : RECORD_LINK := new RECORD_TYPE(TABLE.NUMBER_FIELDS);
  begin
    for I in 1..TABLE.NUMBER_FIELDS loop
      case TABLE.FIELDS(I).DATA_TYPE is
        when INTEGER_FIELD =>
          NEW_RECORD.VALUES(I) := new VALUE_TYPE'(INTEGER_FIELD,0);
        when FLOAT_FIELD =>
          NEW_RECORD.VALUES(I) := new VALUE_TYPE'(FLOAT_FIELD,0.0);
        when STRING_FIELD =>
          NEW_RECORD.VALUES(I) :=
              new VALUE_TYPE'(STRING_FIELD,new STRING'(""));
      end case;
    end loop;
    REC := NEW_RECORD;
  end MAKE_NEW_RECORD;

  procedure INSERT_NEW_RECORD(TABLE : in out TABLE_LINK;
                              REC   : in     RECORD_LINK) is
    LAST_RECORD : RECORD_LINK := TABLE.RECORDS;
  begin
    if LAST_RECORD = null then
      TABLE.RECORDS := REC;
    else
      while LAST_RECORD.NEXT_RECORD /= null loop -- should save last pointer **
        LAST_RECORD := LAST_RECORD.NEXT_RECORD;
      end loop;
      LAST_RECORD.NEXT_RECORD := REC;
    end if;
  end INSERT_NEW_RECORD;

  procedure BUILD_INSERT_LIST(TABLE       : in TABLE_LINK;
                              FIELD_LIST  : in FIELD;
```

```
                                   INSERT_LIST : in out QUERY_NODE) is
begin
  case FIELD_LIST.FIELD_TYPE is
    when OPERATOR =>
      if FIELD_LIST.OPCODE /= O_CAT then
        raise SYNTAX_ERROR;
      end if;
      BUILD_INSERT_LIST(TABLE,FIELD_LIST.DOWN_LINK.ACROSS_LINK,INSERT_LIST);
      BUILD_INSERT_LIST(TABLE,FIELD_LIST.DOWN_LINK,INSERT_LIST);
    when UNQUALIFIED_FIELD =>
      INSERT_LIST := new QUERY_NODE_REC'(INSERT_LIST,
          FIND_FIELD(TABLE,FIELD_LIST.NAME),null);
    when others =>
      raise SYNTAX_ERROR;
  end case;
end BUILD_INSERT_LIST;

procedure INSERT_VALUES(TABLE    : in TABLE_LINK;
                        REC      : in out RECORD_LINK;
                        INTO     : in out QUERY_NODE;
                        LITERALS : in FIELD) is
  FIELD_NUMBER : FIELD_INDEX;
begin
  case LITERALS.FIELD_TYPE is
    when OPERATOR =>
      if LITERALS.OPCODE /= O_AND then
        raise SYNTAX_ERROR;
      end if;
      INSERT_VALUES(TABLE,REC,INTO,LITERALS.DOWN_LINK);
      INSERT_VALUES(TABLE,REC,INTO,LITERALS.DOWN_LINK.ACROSS_LINK);
    when INTEGER_LITERAL | FLOAT_LITERAL | STRING_LITERAL =>
      if INTO = null then
        raise SYNTAX_ERROR;
      end if;
      FIELD_NUMBER := INTO.FIELD;
      if LITERALS.FIELD_TYPE /=
          MATCHING_TYPES(TABLE.FIELDS(FIELD_NUMBER).DATA_TYPE) then
        raise UNIMPLEMENTED_ERROR;
      end if;
      REC.VALUES(FIELD_NUMBER) := CREATE_LITERAL_VALUE(LITERALS);
      INTO := INTO.NEXT_NODE;
    when others =>
      raise SYNTAX_ERROR;
    end case;
end INSERT_VALUES;

procedure ONLY_ONE_TABLE(T : in TABLE) is
begin
  if T.NEXT_LINK /= null then
    raise SYNTAX_ERROR;
  end if;
end ONLY_ONE_TABLE;

procedure DO_INSERT(F : in FIELD) is
  FIELD_LIST   : FIELD := F.ACROSS_LINK;
  INTO_TABLE   : TABLE := FIELD_LIST.TABLE_LINK;
  TABLE_PTR    : TABLE_LINK;
```

```
  VALUE_LIST   : FIELD := F.DOWN_LINK;
  NEW_RECORD   : RECORD_LINK;
  INSERT_LIST  : QUERY_NODE;
begin
  ONLY_ONE_TABLE(INTO_TABLE);
  TABLE_PTR := FIND_TABLE(INTO_TABLE.NAME);
  FIELD_LIST := FIELD_LIST.ACROSS_LINK;
  MAKE_NEW_RECORD(TABLE_PTR,NEW_RECORD);
  if FIELD_LIST = null then
    raise UNIMPLEMENTED_ERROR;
  else
    if VALUE_LIST.FIELD_TYPE = OPERATOR and then
        VALUE_LIST.OPCODE = O_SELECT then
      raise UNIMPLEMENTED_ERROR;
    end if;
    BUILD_INSERT_LIST(TABLE_PTR,FIELD_LIST,INSERT_LIST);
    INSERT_VALUES(TABLE_PTR,NEW_RECORD,INSERT_LIST,VALUE_LIST);
    if INSERT_LIST /= null then
      raise SYNTAX_ERROR;
    end if;
    INSERT_NEW_RECORD(TABLE_PTR,NEW_RECORD);
  end if;
end DO_INSERT;

procedure DO_DELETE(F : in FIELD) is
  WHERE      : FIELD := F.DOWN_LINK;
  FROM       : TABLE := WHERE.TABLE_LINK;
  CURSOR     : CURSOR_TYPE;
  TABLE_PTR  : TABLE_LINK;
  PREVIOUS   : RECORD_LINK;
begin
  ONLY_ONE_TABLE(FROM);
  TABLE_PTR := FIND_TABLE(FROM.NAME);
  CURSOR.CURRENT_RECORD := TABLE_PTR.RECORDS;
  BUILD_WHERE(CURSOR,WHERE.ACROSS_LINK,TABLE_PTR);
  while CURSOR.CURRENT_RECORD /= null and then EQUAL_RECORD(CURSOR) loop
    CURSOR.CURRENT_RECORD := CURSOR.CURRENT_RECORD.NEXT_RECORD;
    TABLE_PTR.RECORDS := CURSOR.CURRENT_RECORD;
  end loop;
  PREVIOUS := CURSOR.CURRENT_RECORD;
  if PREVIOUS /= null then
    while PREVIOUS.NEXT_RECORD /= null loop
      CURSOR.CURRENT_RECORD := PREVIOUS.NEXT_RECORD;
      if EQUAL_RECORD(CURSOR) then
        PREVIOUS.NEXT_RECORD := CURSOR.CURRENT_RECORD.NEXT_RECORD;
      else
        PREVIOUS := CURSOR.CURRENT_RECORD;
      end if;
    end loop;
  end if;
end DO_DELETE;

procedure BUILD_SET_LIST(SET_LIST : in out QUERY_NODE;
                         SET      : in FIELD;
                         WHAT     : in TABLE_LINK) is
  FIELD_NUMBER : FIELD_INDEX;
  TARGET_TYPE  : DATABASE_FIELD_TYPE;
```

```
    LEFT,
    RIGHT          : FIELD;
  begin
    if SET.FIELD_TYPE /= OPERATOR then
      raise SYNTAX_ERROR;
    end if;
    LEFT := SET.DOWN_LINK; RIGHT := LEFT.ACROSS_LINK;
    case SET.OPCODE is
      when O_CAT =>
        BUILD_SET_LIST(SET_LIST,RIGHT,WHAT);
        BUILD_SET_LIST(SET_LIST,LEFT,WHAT);
      when O_EQ =>
        if LEFT.FIELD_TYPE /= UNQUALIFIED_FIELD then
          raise SYNTAX_ERROR;
        end if;
        FIELD_NUMBER := FIND_FIELD(WHAT,LEFT.NAME);
        TARGET_TYPE := WHAT.FIELDS(FIELD_NUMBER).DATA_TYPE;
        if RIGHT.FIELD_TYPE /=
            MATCHING_TYPES(TARGET_TYPE) then
          raise UNIMPLEMENTED_ERROR;
        end if;
        SET_LIST := new QUERY_NODE_REC'(SET_LIST,FIELD_NUMBER,
          CREATE_LITERAL_VALUE(RIGHT));
      when others =>
        raise SYNTAX_ERROR;
    end case;
  end BUILD_SET_LIST;

  procedure DO_UPDATE(F : in FIELD) is
    FROM      : TABLE := F.DOWN_LINK.TABLE_LINK;
    SET       : FIELD := F.DOWN_LINK.ACROSS_LINK;
    WHERE     : FIELD := SET.ACROSS_LINK;
    TABLE_PTR : TABLE_LINK;
    SET_LIST,
    SET_NOW   : QUERY_NODE;
    CURSOR    : CURSOR_TYPE;
  begin
    ONLY_ONE_TABLE(FROM);
    TABLE_PTR := FIND_TABLE(FROM.NAME);
    CURSOR.CURRENT_RECORD := TABLE_PTR.RECORDS;
    BUILD_WHERE(CURSOR,WHERE,TABLE_PTR);
    BUILD_SET_LIST(SET_LIST,SET,TABLE_PTR);
    loop
      NEXT_RECORD(CURSOR);
      SET_NOW := SET_LIST;
      while SET_NOW /= null loop
        CURSOR.CURRENT_RECORD.VALUES(SET_NOW.FIELD) := SET_NOW.VALUE;
        SET_NOW := SET_NOW.NEXT_NODE;
      end loop;
    end loop;
  exception
    when DONE_ERROR =>
      return;
  end DO_UPDATE;

  procedure EXECUTE(F : in FIELD) is
  begin
```

```
      case F.OPCODE is
        when O_INSERT =>
          DO_INSERT(F);
        when O_DELETE =>
          DO_DELETE(F);
        when O_UPDATE =>
          DO_UPDATE(F);
        when others =>
          raise SYNTAX_ERROR;
      end case;
    exception
      when CONSTRAINT_ERROR =>
        raise SYNTAX_ERROR;
    end EXECUTE;

    procedure LIST(F : in FIELD) is
    begin
      null;
    end LIST;

  end PROGRAM_FUNCTIONS;
```

```
with TEXT_INPUT, TEXT_IO, TEXT_PRINT;
  use TEXT_INPUT, TEXT_IO, TEXT_PRINT;

separate(SQL_DEFINITIONS.SQL_FUNCTIONS)
package body BULK_FUNCTIONS is

  type TABLE_LIST_REC;

  type TABLE_LIST_LINK is access TABLE_LIST_REC;

  type TABLE_LIST_REC is
    record
      NEXT_TABLE : TABLE_LIST_LINK;
      TABLE      : TABLE_LINK;
    end record;

  type FIELD_LINK is access FIELD_TYPE;

  type FIELD_LIST_REC;

  type FIELD_LIST_LINK is access FIELD_LIST_REC;

  type FIELD_LIST_REC is
    record
      NEXT_FIELD : FIELD_LIST_LINK;
      FIELD      : FIELD_LINK;
    end record;

  function CHECK_FIELD_LIST(BUFFER     : BUFFER_TYPE;
                            FIELD_LIST : FIELD_LIST_LINK; -- return -> last one
                            NAME       : FIELD_NAME) return FIELD_LIST_LINK is
    FIELD : FIELD_LIST_LINK := FIELD_LIST;
  begin
    loop
      if NAME.all = FIELD.FIELD.NAME.all then
        CARD_ERROR(BUFFER,"DBLOAD - Duplicate FIELD name");
      end if;
      exit when FIELD.NEXT_FIELD = null;
      FIELD := FIELD.NEXT_FIELD;
    end loop;
    return FIELD;
  end CHECK_FIELD_LIST;

  function CHECK_TABLE_LIST(BUFFER     : BUFFER_TYPE;
                            TABLE_LIST : TABLE_LIST_LINK; -- return -> last one
                            NAME       : TABLE_NAME) return TABLE_LIST_LINK is
    TABLE : TABLE_LIST_LINK := TABLE_LIST;
  begin
    loop
      if NAME.all = TABLE.TABLE.NAME.all then
        CARD_ERROR(BUFFER,"DBLOAD - Duplicate TABLE name");
      end if;
      exit when TABLE.NEXT_TABLE = null;
      TABLE := TABLE.NEXT_TABLE;
    end loop;
    return TABLE;
  end CHECK_TABLE_LIST;
```

```
function COMBINE_FIELDS(TABLE       : TABLE_NAME;
                        FIRST_FIELD : FIELD_LIST_LINK) return TABLE_LINK is
  F : FIELD_LIST_LINK := FIRST_FIELD;
  T : TABLE_LINK;
  C : EXTENDED_FIELD_INDEX := 0;
begin
  while F /= null loop
    C := C + 1;
    F := F.NEXT_FIELD;
  end loop;
  T := new TABLE_TYPE(C);
  T.NAME := TABLE;
  F := FIRST_FIELD;
  for I in 1..C loop
    T.FIELDS(I) := F.FIELD.all;
    F := F.NEXT_FIELD;
  end loop;
  return T;
end COMBINE_FIELDS;

function COMBINE_TABLES(FIRST_TABLE : TABLE_LIST_LINK) return DATABASE_TYPE
    is
  D : DATABASE_TYPE;
  T : TABLE_LIST_LINK := FIRST_TABLE;
  C : EXTENDED_TABLE_INDEX := 0;
begin
  while T /= null loop
    C := C + 1;
    T := T.NEXT_TABLE;
  end loop;
  D := new TABLE_ARRAY(1..C);
  T := FIRST_TABLE;
  for I in 1..C loop
    D(I) := T.TABLE;
    T := T.NEXT_TABLE;
  end loop;
  return D;
end COMBINE_TABLES;

procedure GET_DATA(BUFFER : in out BUFFER_TYPE;
                   IDENT  : in out STRING;
                   LAST   : in out POSITIVE;
                   TABLE  : in out TABLE_LINK) is
  LAST_RECORD  : RECORD_LINK := new RECORD_TYPE(0);
begin
  TABLE.RECORDS := LAST_RECORD;
  loop
    exit when IDENT(1..LAST) /= "DATA";
    LAST_RECORD.NEXT_RECORD := new RECORD_TYPE(TABLE.NUMBER_FIELDS);
    LAST_RECORD := LAST_RECORD.NEXT_RECORD;
    begin
      for I in 1..TABLE.NUMBER_FIELDS loop
        case TABLE.FIELDS(I).DATA_TYPE is
          when INTEGER_FIELD =>
            LAST_RECORD.VALUES(I) :=
              new VALUE_TYPE'(INTEGER_FIELD,IN_INTEGER(BUFFER));
          when FLOAT_FIELD =>
```

```
              LAST_RECORD.VALUES(I) :=
                new VALUE_TYPE'(FLOAT_FIELD, IN_FLOAT(BUFFER));
            when STRING_FIELD =>
              LAST_RECORD.VALUES(I) :=
                new VALUE_TYPE'(STRING_FIELD, IN_STRING(BUFFER));
              if LAST_RECORD.VALUES(I).STRING_VALUE'LENGTH >
                  TABLE.FIELDS(I).SIZE then
                CARD_ERROR(BUFFER, "DBLOAD - STRING longer than declaration");
              end if;
          end case;
        end loop;
      exception
        when END_ERROR =>
          CARD_ERROR(BUFFER, "DBLOAD - end of file before all DATA read");
        when others =>
          CARD_ERROR(BUFFER, "DBLOAD - improper format on data");
      end;
      IN_IDENT(BUFFER, IDENT, LAST);
    end loop;
    TABLE.RECORDS := TABLE.RECORDS.NEXT_RECORD;
  exception
    when END_ERROR =>
      TABLE.RECORDS := TABLE.RECORDS.NEXT_RECORD;
  end GET_DATA;

  procedure GET_FIELDS(BUFFER : in out BUFFER_TYPE;
                       IDENT  : in out STRING;
                       LAST   : in out POSITIVE;
                       FIELD1 : out    FIELD_LIST_LINK) is
    FIELD_LIST : FIELD_LIST_LINK := new FIELD_LIST_REC'(null,
        new FIELD_TYPE'(new FIELD_NAME_STRING'(""),STRING_FIELD,1));
    LAST_FIELD : FIELD_LIST_LINK;
    FLD_NAME   : FIELD_NAME;
    TYPE_FIELD : DATABASE_FIELD_TYPE;
  begin
    loop
      exit when IDENT(1..LAST) /= "FIELD";
      begin
        IN_IDENT(BUFFER, IDENT, LAST);
        FLD_NAME := new FIELD_NAME_STRING'(FIELD_NAME_STRING(IDENT(1..LAST)));
        LAST_FIELD := CHECK_FIELD_LIST(BUFFER,FIELD_LIST,FLD_NAME);
        IN_IDENT(BUFFER, IDENT, LAST);
        TYPE_FIELD := DATABASE_FIELD_TYPE'VALUE(IDENT(1..LAST) & "_FIELD");
        LAST_FIELD.NEXT_FIELD := new FIELD_LIST_REC'(null,
            new FIELD_TYPE'(FLD_NAME,TYPE_FIELD,1));
        LAST_FIELD.NEXT_FIELD.FIELD.SIZE := IN_INTEGER(BUFFER);
      exception
        when END_ERROR =>
          CARD_ERROR(BUFFER,
              "DBLOAD - premature end of file in FIELD description");
        when others =>
          CARD_ERROR(BUFFER, "DBLOAD - invalid field description");
      end;
      IN_IDENT(BUFFER, IDENT, LAST);
    end loop;
    FIELD1 := FIELD_LIST.NEXT_FIELD;
  exception
```

```
    when END_ERROR =>
      FIELD1 := FIELD_LIST.NEXT_FIELD;
  end GET_FIELDS;

  function LOAD_DATABASE(FILE_NAME : in STRING) return DATABASE_TYPE is
    BUFFER      : BUFFER_TYPE := MAKE_BUFFER(100);
    IDENT       : STRING(1..100);
    LAST        : NATURAL;
    TABLE_LIST  : TABLE_LIST_LINK := new TABLE_LIST_REC'(null,
        new TABLE_TYPE'(0,new TABLE_NAME_STRING'(""),null,
        (1..0 => FIELD_TYPE'(new FIELD_NAME_STRING'(""),STRING_FIELD,1))));
    FIELD_LIST  : FIELD_LIST_LINK;
    LAST_TABLE  : TABLE_LIST_LINK;
    TBL_NAME    : TABLE_NAME;
  begin
    OPEN_INPUT(BUFFER,IN_FILE,FILE_NAME);
    IN_IDENT(BUFFER,IDENT,LAST);
    while not END_OF_FILE(BUFFER) loop
      exit when IDENT(1..LAST) = "END"; -- exceptions are not propagating right
      if IDENT(1..LAST) /= "TABLE" then
        CARD_ERROR(BUFFER,"DBLOAD - TABLE card expected, not found");
      end if;
      begin
        IN_IDENT(BUFFER,IDENT,LAST);
      exception
        when others =>
          CARD_ERROR(BUFFER,"DBLOAD - invalid TABLE card");
      end;
      TBL_NAME := new TABLE_NAME_STRING'(TABLE_NAME_STRING(IDENT(1..LAST)));
      LAST_TABLE := CHECK_TABLE_LIST(BUFFER,TABLE_LIST,TBL_NAME);
      IN_IDENT(BUFFER,IDENT,LAST);
      GET_FIELDS(BUFFER,IDENT,LAST,FIELD_LIST);
      LAST_TABLE.NEXT_TABLE := new TABLE_LIST_REC'(null,
        COMBINE_FIELDS(TBL_NAME,FIELD_LIST));
      GET_DATA(BUFFER,IDENT,LAST,LAST_TABLE.NEXT_TABLE.TABLE);
    end loop;
    CLOSE_INPUT(BUFFER);
    return COMBINE_TABLES(TABLE_LIST.NEXT_TABLE);
  end LOAD_DATABASE;

  procedure SAVE_DATA(FILE  : in FILE_TYPE;
                      TABLE : in TABLE_LINK;
                      REC   : in RECORD_LINK) is
  begin
    PRINT(FILE,"DATA");
    for I in 1..TABLE.NUMBER_FIELDS loop
      PRINT(FILE," ");
      case TABLE.FIELDS(I).DATA_TYPE is
        when INTEGER_FIELD =>
          PRINT(FILE,REC.VALUES(I).INTEGER_VALUE,NO_BREAK);
        when FLOAT_FIELD =>
          PRINT(FILE,REC.VALUES(I).FLOAT_VALUE,NO_BREAK);
        when STRING_FIELD =>
          PRINT(FILE,"""" & REC.VALUES(I).STRING_VALUE.all & """",NO_BREAK);
      end case;
    end loop;
    PRINT_LINE(FILE);
```

```
  end SAVE_DATA;

  procedure SAVE_FIELDS(FILE : in FILE_TYPE; TABLE : in TABLE_LINK) is
    FIELD : FIELD_TYPE;
  begin
    for I in 1..TABLE.NUMBER_FIELDS loop
      FIELD := TABLE.FIELDS(I);
      PRINT(FILE,"FIELD " & STRING(FIELD.NAME.all) & " ");
      case FIELD.DATA_TYPE is
        when INTEGER_FIELD =>
          PRINT(FILE,"INTEGER ",NO_BREAK);
        when FLOAT_FIELD =>
          PRINT(FILE,"FLOAT ",NO_BREAK);
        when STRING_FIELD =>
          PRINT(FILE,"STRING ",NO_BREAK);
      end case;
      PRINT(FILE,FIELD.SIZE); PRINT_LINE(FILE);
    end loop;
  end SAVE_FIELDS;

  procedure SAVE_DATABASE(FILE_NAME : in STRING; DATABASE : in DATABASE_TYPE)
      is
    FILE  : FILE_TYPE;
    L     : LINE_TYPE;
    TABLE : TABLE_LINK;
    REC   : RECORD_LINK;
  begin
    CREATE(FILE,OUT_FILE,FILE_NAME); CREATE_LINE(L,79); SET_LINE(L);
    for I in 1..DATABASE'LAST loop
      BLANK_LINE(FILE);
      TABLE := DATABASE(I);
      PRINT(FILE,"TABLE " & STRING(TABLE.NAME.all));
      PRINT_LINE(FILE); BLANK_LINE(FILE);
      SAVE_FIELDS(FILE,TABLE);
      REC := TABLE.RECORDS;
      if REC /= null then
        BLANK_LINE(FILE);
        while REC /= null loop
          SAVE_DATA(FILE,TABLE,REC);
          REC := REC.NEXT_RECORD;
        end loop;
      end if;
    end loop;
    BLANK_LINE(FILE); PRINT(FILE,"END"); PRINT_LINE(FILE);
    PRINT(FILE,"END"); PRINT_LINE(FILE);
    CLOSE(FILE);
  end SAVE_DATABASE;

end BULK_FUNCTIONS;
```

```
with TEXT_PRINT;
  use TEXT_PRINT;

separate(SQL_DEFINITIONS.SQL_FUNCTIONS)
package body SHOW_PACKAGE is

  type TABLE_LIST_REC;

  type TABLE_LIST is access TABLE_LIST_REC;

  type TABLE_LIST_REC is
    record
      NAME,
      PRINT          : TABLE_NAME;
      VERSION_LINK,
      NAME_LINK      : TABLE_LIST;
    end record;

  type PRECEDENCE_TYPE is range 1..10; -- SQL, not Ada, operator precedence

  type CLAUSE_NAME_TYPE is new STRING(1..7);

  procedure SHOWR(F : in FIELD);

  INDENT        : INTEGER;
  TABLE_TABLE   : TABLE_LIST;
  DOING_SET     : BOOLEAN := FALSE;
  INITIAL_TABLE : constant TABLE_LIST := new TABLE_LIST_REC'(
                              new TABLE_NAME_STRING'(""),null,null,null);

  PRECEDENCE : constant array(OPERATOR_TYPE) of PRECEDENCE_TYPE := (
    O_SELECT | O_INSERT | O_DELETE | O_UPDATE | O_SUM | O_AVG | O_MAX |
        O_MIN | O_COUNT | O_DESC | O_CAT                              => 10,
    O_ABS                                                             => 9,
    O_POWER                                                           => 8,
    O_TIMES | O_DIV | O_MOD | O_REM                                   => 7,
    O_UNARY_PLUS | O_UNARY_MINUS                                      => 6,
    O_PLUS | O_MINUS                                                  => 5,
    O_EQ | O_NE | O_LT | O_LE | O_GT | O_GE | O_LIKE | O_IN | O_EXISTS => 4,
    O_NOT                                                             => 3,
    O_AND                                                             => 2,
    O_OR | O_XOR                                                      => 1);

  OPERATOR_NAME : constant array(OPERATOR_TYPE) of STRING_LINK := (
    new STRING'("SELECT"), new STRING'("INSERT"), new STRING'("DELETE"),
    new STRING'("UPDATE"), new STRING'("LIKE"),   new STRING'("SUM"),
    new STRING'("AVG"),    new STRING'("MAX"),    new STRING'("MIN"),
    new STRING'("COUNT"),  new STRING'("IN"),     new STRING'("EXISTS"),
    new STRING'("DESC"),   new STRING'("AND"),    new STRING'("OR"),
    new STRING'("XOR"),    new STRING'("="),      new STRING'("/="),
    new STRING'("<"),      new STRING'("<="),     new STRING'(">"),
    new STRING'(">="),     new STRING'("+"),      new STRING'("-"),
    new STRING'(","),      new STRING'("+"),      new STRING'("-"),
    new STRING'("*"),      new STRING'("/"),      new STRING'("MOD"),
    new STRING'("REM"),    new STRING'("**"),   new STRING'("ABS"),
    new STRING'("NOT") );
```

```
CLAUSE_NAME : constant array(1..4) of CLAUSE_NAME_TYPE :=
  ("WHERE  ", "GROUP  ", "HAVING ", "ORDER  ");

HAS_BY : constant array(1..4) of BOOLEAN := (FALSE, TRUE, FALSE, TRUE);

SIX_BLANKS : constant STRING := "      ";

procedure ENTER_NEW_TABLE(T : TABLE_NAME) is
  NAME_ENTRY    : TABLE_LIST := TABLE_TABLE;
  VERSION_ENTRY : TABLE_LIST;
begin
  loop
    if NAME_ENTRY.NAME.all = T.all then
      VERSION_ENTRY := NAME_ENTRY;
      loop
        if VERSION_ENTRY.NAME = T then
          return;
        end if;
        exit when VERSION_ENTRY.VERSION_LINK = null;
        VERSION_ENTRY := VERSION_ENTRY.VERSION_LINK;
      end loop;
      VERSION_ENTRY.VERSION_LINK := new TABLE_LIST_REC'(T,T,null,null);
      return;
    end if;
    exit when NAME_ENTRY.NAME_LINK = null;
    NAME_ENTRY := NAME_ENTRY.NAME_LINK;
  end loop;
  NAME_ENTRY.NAME_LINK := new TABLE_LIST_REC'(T,T,null,null);
end ENTER_NEW_TABLE;

procedure CREATE_TABLE_TABLE(F : in FIELD) is
  G : FIELD;
  T : TABLE;
begin
  case F.FIELD_TYPE is
    when OPERATOR =>
      G := F.DOWN_LINK;
      while G /= null loop
        CREATE_TABLE_TABLE(G);
        G := G.ACROSS_LINK;
      end loop;
    when QUALIFIED_FIELD =>
      ENTER_NEW_TABLE(F.RELATION);
    when FROM_LIST =>
      T := F.TABLE_LINK;
      while T /= null loop
        ENTER_NEW_TABLE(T.NAME);
        T := T.NEXT_LINK;
      end loop;
    when others =>
      null;
  end case;
end CREATE_TABLE_TABLE;

procedure FINALIZE_TABLE_TABLE is
  NAME_ENTRY : TABLE_LIST := TABLE_TABLE;
  VERSION_ENTRY,NEXT_NAME,NEXT_VERSION : TABLE_LIST;
```

```
  VERSION_NUMBER,NAME_LENGTH : INTEGER;
begin
  while NAME_ENTRY /= null loop
    if NAME_ENTRY.VERSION_LINK /= null then
      VERSION_NUMBER := 1;
      VERSION_ENTRY := NAME_ENTRY;
      NAME_LENGTH := VERSION_ENTRY.NAME'LENGTH;
      while VERSION_ENTRY /= null loop
        VERSION_ENTRY.PRINT := new TABLE_NAME_STRING'(
          TABLE_NAME_STRING(
            STRING(VERSION_ENTRY.NAME.all) &
              INTEGER'IMAGE(VERSION_NUMBER) & ")" ) );
        VERSION_ENTRY.PRINT(NAME_LENGTH+1) := '(';
        VERSION_NUMBER := VERSION_NUMBER + 1;
        NEXT_NAME := NAME_ENTRY.NAME_LINK;
        NEXT_VERSION := VERSION_ENTRY.VERSION_LINK;
        NAME_ENTRY.NAME_LINK := VERSION_ENTRY;
        VERSION_ENTRY.NAME_LINK := NEXT_NAME;
        NAME_ENTRY := VERSION_ENTRY;
        VERSION_ENTRY := NEXT_VERSION;
      end loop;
    end if;
    NAME_ENTRY := NAME_ENTRY.NAME_LINK;
  end loop;
end FINALIZE_TABLE_TABLE;

procedure SHOW_TABLE_NAME(NAME : in TABLE_NAME) is
  T : TABLE_LIST := TABLE_TABLE;
begin
  loop
    if NAME = T.NAME then
      PRINT(STRING(T.PRINT.all),NO_BREAK);
      return;
    end if;
    T := T.NAME_LINK;
  end loop;
end SHOW_TABLE_NAME;

procedure SHOW_SELECT(F : in FIELD) is
  CLAUSE : FIELD;
  T      : TABLE;
begin
  INDENT := INDENT + 7;
  if INDENT > 0 then
    SET_INDENT(INDENT-1); PRINT_LINE; PRINT("("); SET_INDENT(INDENT);
  else
    SET_INDENT(INDENT); PRINT_LINE;
  end if;
  PRINT("SELECT "); CLAUSE := F.DOWN_LINK; SHOWR(CLAUSE);
  CLAUSE := CLAUSE.ACROSS_LINK; T := CLAUSE.TABLE_LINK;
  if T /= null then
    PRINT_LINE; PRINT("FROM   ");
    loop
      SHOW_TABLE_NAME(T.NAME); T := T.NEXT_LINK;
      exit when T = null;
      PRINT(", ");
    end loop;
```

```
    end if;
    for I in 1..4 loop
      CLAUSE := CLAUSE.ACROSS_LINK;
      if CLAUSE.FIELD_TYPE /= EMPTY then
        PRINT_LINE; PRINT(STRING(CLAUSE_NAME(I)));
        if HAS_BY(I) then
          PRINT("BY ");
        end if;
        SHOWR(CLAUSE);
      end if;
    end loop;
    INDENT := INDENT - 7;
    if INDENT >= 0 then
      PRINT(")"); SET_INDENT(INDENT);
    end if;
  end SHOW_SELECT;

  procedure START_STATEMENT is
  begin
    INDENT := INDENT + 7; SET_INDENT(INDENT); PRINT_LINE;
  end START_STATEMENT;

  procedure SHOW_INSERT(F : in FIELD) is
    CLAUSE : FIELD;
  begin
    START_STATEMENT; PRINT("INSERT INTO ");
    CLAUSE := F.ACROSS_LINK;
    SHOW_TABLE_NAME(CLAUSE.TABLE_LINK.NAME);
    if CLAUSE.ACROSS_LINK /= null then
      PRINT(" ( "); SHOWR(CLAUSE.ACROSS_LINK); PRINT(" )");
    end if;
    CLAUSE := F.DOWN_LINK;
    if CLAUSE.FIELD_TYPE = OPERATOR and then CLAUSE.OPCODE = O_SELECT then
      SHOW_SELECT(CLAUSE);
    else
      START_STATEMENT;
      PRINT("VALUES ("); SHOWR(CLAUSE); PRINT(")");
      INDENT := INDENT - 7; SET_INDENT(INDENT);
    end if;
    INDENT := INDENT - 7;
  end SHOW_INSERT;

  procedure SHOW_WHERE(F : in FIELD) is
  begin
    if F.FIELD_TYPE /= EMPTY then
      PRINT_LINE; PRINT("WHERE  "); SHOWR(F);
    end if;
  end SHOW_WHERE;

  procedure SHOW_DELETE(F : in FIELD) is
    CLAUSE : FIELD;
  begin
    START_STATEMENT; PRINT("DELETE");
    CLAUSE := F.DOWN_LINK;
    if CLAUSE.TABLE_LINK /= null then
      PRINT_LINE; PRINT("FROM   "); SHOW_TABLE_NAME(CLAUSE.TABLE_LINK.NAME);
    end if;
```

```
    SHOW_WHERE(CLAUSE.ACROSS_LINK);
    INDENT := INDENT - 7;
  end SHOW_DELETE;

  procedure SHOW_UPDATE(F : in FIELD) is
    CLAUSE : FIELD;
  begin
    START_STATEMENT; PRINT("UPDATE ");
    CLAUSE := F.DOWN_LINK;
    if CLAUSE.TABLE_LINK /= null then
      SHOW_TABLE_NAME(CLAUSE.TABLE_LINK.NAME);
    end if;
    PRINT_LINE; PRINT("SET    "); INDENT := INDENT + 7; SET_INDENT(INDENT);
    CLAUSE := CLAUSE.ACROSS_LINK;
    DOING_SET := TRUE; SHOWR(CLAUSE); DOING_SET := FALSE;
    INDENT := INDENT - 7; SET_INDENT(INDENT);
    SHOW_WHERE(CLAUSE.ACROSS_LINK);
    INDENT := INDENT - 7;
  end SHOW_UPDATE;

  procedure SHOW_PRECEDENCE(UPPER_PRECEDENCE : in PRECEDENCE_TYPE;
                            OPERAND          : in FIELD) is
  begin
    if OPERAND.FIELD_TYPE = OPERATOR and then
        PRECEDENCE(OPERAND.OPCODE) < UPPER_PRECEDENCE and then
        OPERAND.DOWN_LINK.ACROSS_LINK /= null then
      PRINT("( "); SHOWR(OPERAND); PRINT(" )");
    else
      SHOWR(OPERAND);
    end if;
  end SHOW_PRECEDENCE;

  procedure SHOW_MARGIN(F : in FIELD) is
  begin
    SHOW_PRECEDENCE(PRECEDENCE(F.OPCODE),F.DOWN_LINK); PRINT_LINE;
    PRINT(OPERATOR_NAME(F.OPCODE).all &
        SIX_BLANKS(OPERATOR_NAME(F.OPCODE)'LENGTH..6));
    SHOW_PRECEDENCE(PRECEDENCE(F.OPCODE),F.DOWN_LINK.ACROSS_LINK);
  end SHOW_MARGIN;

  procedure SHOW_LIST(F : in FIELD) is
  begin
    SHOWR(F.DOWN_LINK); PRINT(", ");
    if DOING_SET then
      PRINT_LINE;
    end if;
    SHOWR(F.DOWN_LINK.ACROSS_LINK);
  end SHOW_LIST;

  procedure SHOW_OPERATOR(F : in FIELD) is
  begin
    case F.OPCODE is
      when O_SELECT =>
        SHOW_SELECT(F);
      when O_INSERT =>
        SHOW_INSERT(F);
      when O_DELETE =>
```

```
        SHOW_DELETE(F);
      when O_UPDATE =>
        SHOW_UPDATE(F);
      when O_SUM | O_AVG | O_MAX | O_MIN | O_COUNT =>
        PRINT(OPERATOR_NAME(F.OPCODE).all & "("); SHOWR(F.DOWN_LINK);
        PRINT(")");
      when O_DESC =>
        SHOWR(F.DOWN_LINK); PRINT(" DESC");
      when O_IN =>
        SHOW_PRECEDENCE(PRECEDENCE(O_IN),F.DOWN_LINK); PRINT(" IN ");
        if F.DOWN_LINK.ACROSS_LINK.FIELD_TYPE /= OPERATOR or else
             F.DOWN_LINK.ACROSS_LINK.OPCODE /= O_SELECT then
          PRINT("( "); SHOWR(F.DOWN_LINK.ACROSS_LINK); PRINT(" )");
        else
          SHOWR(F.DOWN_LINK.ACROSS_LINK);
        end if;
      when O_LIKE | O_EQ | O_NE | O_LT | O_LE | O_GT | O_GE | O_PLUS |
           O_MINUS | O_TIMES | O_DIV | O_MOD | O_REM | O_POWER =>
        SHOW_PRECEDENCE(PRECEDENCE(F.OPCODE),F.DOWN_LINK);
        PRINT(" " & OPERATOR_NAME(F.OPCODE).all & " ");
        SHOW_PRECEDENCE(PRECEDENCE(F.OPCODE),F.DOWN_LINK.ACROSS_LINK);
      when O_EXISTS | O_UNARY_PLUS | O_UNARY_MINUS | O_ABS | O_NOT =>
        PRINT(OPERATOR_NAME(F.OPCODE).all & " ");
        SHOW_PRECEDENCE(PRECEDENCE(F.OPCODE),F.DOWN_LINK);
      when O_AND | O_OR =>
        if F.DOWN_LINK.FIELD_TYPE = OPERATOR and then
            F.DOWN_LINK.ACROSS_LINK.FIELD_TYPE = OPERATOR then
          SHOW_MARGIN(F);
        else
          SHOW_LIST(F);
        end if;
      when O_XOR =>
        SHOW_MARGIN(F);
      when O_CAT =>
        SHOW_LIST(F);
    end case;
  end SHOW_OPERATOR;

  procedure SHOWR(F : in FIELD) is
    T : TABLE_LIST;
  begin
    case F.FIELD_TYPE is
      when OPERATOR =>
        SHOW_OPERATOR(F);
      when INTEGER_LITERAL =>
        PRINT(F.INTEGER_VALUE);
      when STRING_LITERAL =>
        PRINT("'" & F.STRING_VALUE.all & "'");
      when FLOAT_LITERAL =>
        PRINT(F.FLOAT_VALUE);
      when QUALIFIED_FIELD =>
        SHOW_TABLE_NAME(F.RELATION);
        PRINT("." & STRING(F.NAME.all));
      when UNQUALIFIED_FIELD =>
        PRINT(STRING(F.NAME.all));
      when FROM_LIST | EMPTY =>
        null;
```

```
    end case;
  end SHOWR;

  procedure SHOW(F : in FIELD) is
  begin
    INDENT := -7;
    SET_CONTINUATION_INDENT(7);
    BLANK_LINE;
    INITIAL_TABLE.NAME_LINK := null;
    INITIAL_TABLE.VERSION_LINK := null;
    TABLE_TABLE := INITIAL_TABLE;
    CREATE_TABLE_TABLE(F);
    if F.ACROSS_LINK /= null then
      CREATE_TABLE_TABLE(F.ACROSS_LINK);
    end if;
    FINALIZE_TABLE_TABLE;
    SHOWR(F);
    PRINT_LINE;
  end SHOW;

end SHOW_PACKAGE;
```

```
with DAMES_DDL, READ_DDL, SHOW_DDL, SIMPLE_DDL, TEXT_IO, TEXT_PRINT,
    TOKEN_INPUT;
  use DAMES_DDL, READ_DDL, SHOW_DDL, SIMPLE_DDL, TEXT_IO, TEXT_PRINT,
      TOKEN_INPUT;

procedure MAIN is

  LINE         : LINE_TYPE;
  PACKAGE_NAME : STRING(1..80);
  LAST         : NATURAL;

  procedure PRINT_RULE is
  begin
    PRINT("-------------------------------------" &
          "-------------------------------------"); PRINT_LINE;
  end PRINT_RULE;

begin
  SET_STREAM(CREATE_STREAM(80)); OPEN_INPUT("BOATS.ADA");
  CREATE_LINE(LINE,79); SET_LINE(LINE);
  SCAN_DDL(PACKAGE_NAME,LAST);
  DISPLAY_DDL(PACKAGE_NAME(1..LAST)); PRINT_RULE;
  GENERATE_SIMPLE_DDL; PRINT_RULE;
  GENERATE_DAMES_DDL;
  CLOSE_INPUT;
end MAIN;
```

```
TABLE PARCELS

FIELD APN STRING 9
FIELD ROAD STRING 7
FIELD OWNER STRING 20
FIELD IMPROVED STRING 1
FIELD LAST_ENTRY INTEGER 3
FIELD BALANCE FLOAT 7

DATA "93-293-02" "MILL"  "P.J.DEAN" "Y" 17 120.00
DATA "93-282-55" "CREEK" "I.J.KING" "Y"  1 120.00

TABLE OWNERS

FIELD OWNER STRING 20
FIELD ADDRESS STRING 40
FIELD PHONE STRING 12

DATA "P.J.DEAN" "23 THE ALBANY"    "441-296-2015"
DATA "I.J.KING" "15666 CREEK ROAD" ""

TABLE PARCEL_ACCOUNTS

FIELD APN STRING 9
FIELD ENTRY INTEGER 3
FIELD DATE STRING 6
FIELD DESCRIPTION STRING 20
FIELD TYPE STRING 6
FIELD AMOUNT FLOAT 7
FIELD BALANCE FLOAT 7

DATA "93-293-02" 17 "821016" "DAMAGE FEE"  "CHARGE" 500.00 560.00
DATA "93-282-55"  1 "800101" "DUES80"      "CHARGE" 120.00 120.00

DATA "93-281-24" 31 "820107" "DUES82"      "CHARGE" 120.00 120.00
DATA "93-281-24" 32 "820107" "SA7"         "CHARGE" 240.00 360.00
DATA "93-281-24" 33 "820408" "DUES82"      "CREDIT" 120.00 240.00
DATA "93-281-24" 34 "820408" "SA7"         "CREDIT" 240.00   0.00
DATA "93-281-24" 35 "820809" "SA8"         "CHARGE" 115 00 115.00
DATA "93-281-24" 36 "821105" "SA10"        "CHARGE"  72.00 187.00
DATA "93-281-24" 37 "821231" "PENALTY82"   "CHARGE"  37.40 224.40
DATA "93-282-54" 16 "820107" "DUES82"      "CHARGE"  60.00 370.00
DATA "93-282-54" 17 "820107" "SA7"         "CHARGE" 240.00 610.00
DATA "93-282-54" 18 "821231" "PENALTY82"   "CHARGE" 122.00 732.00
DATA "93-282-55" 40 "820107" "DUES82"      "CHARGE" 120.00 120.00
DATA "93-282-55" 41 "820107" "SA7"         "CHARGE" 240.00 360.00
DATA "93-282-55" 42 "820203" "DUES82"      "CREDIT" 120.00 240.00
DATA "93-282-55" 43 "820203" "SA7"         "CREDIT" 240.00   0.00
DATA "92-291-19"  7 "820107" "DUES82"      "CHARGE"  50.00 190.00
DATA "92-291-19"  8 "820107" "SA7"         "CHARGE" 240.00 430.00
DATA "92-291-19"  9 "820809" "SA8"         "CHARGE" 115.00 545.00
DATA "92-291-19" 10 "821105" "SA10"        "CHARGE"  72.00 617.00
DATA "92-291-19" 11 "821123" "SA3"         "CHARGE"  42.50 659.50
DATA "92-291-19" 12 "821231" "PENALTY82"   "CHARGE" 131.90 791.40
DATA "92-291-44" 22 "820107" "DUES82"      "CHARGE" 120.00 120.00
DATA "92-291-44" 23 "821124" "DUES82"      "CREDIT"  60.00  60.00
DATA "92-291-44" 24 "821212" "DUES82"      "CREDIT"  60.00  60.00
```

```
DATA "92-293-02"  4 "820107" "DUES82"     "CHARGE"  60.00  60.00
DATA "92-293-02"  5 "820309" "DUES82"     "CREDIT"  60.00   0.00
DATA "92-293-02"  6 "821105" "SA10"       "CHARGE"  72.00  72.00
DATA "92-293-02"  7 "821119" "SA10"       "CREDIT"  72.00   0.00

TABLE SPECIAL_ASSESSMENTS

FIELD SAN INTEGER 3
FIELD ROAD STRING 7
FIELD DATE STRING 6
FIELD TOTAL FLOAT 7
FIELD PER_PARCEL FLOAT 7
FIELD EXPLANATION STRING 10
FIELD PAYEE STRING 20

DATA 3 "CREEK" "810522" 2460.00 205.00 "GRADING" "ROAD FIXERS, INC."

TABLE GENERAL_LEDGER

FIELD ENTRY INTEGER 3
FIELD DATE STRING 6
FIELD DESCRIPTION STRING 20
FIELD TYPE STRING 6
FIELD PARTY STRING 10
FIELD AMOUNT FLOAT 7
FIELD BALANCE FLOAT 7

DATA 724 "820720" "DUES82" "CREDIT" "93-291-44" 120.00 6095.40

TABLE REDWOOD_LEDGER

FIELD ENTRY INTEGER 3
FIELD DATE STRING 6
FIELD DESCRIPTION STRING 20
FIELD TYPE STRING 6
FIELD PARTY STRING 10
FIELD AMOUNT FLOAT 7
FIELD BALANCE FLOAT 7

TABLE CREEK_LEDGER

FIELD ENTRY INTEGER 3
FIELD DATE STRING 6
FIELD DESCRIPTION STRING 20
FIELD TYPE STRING 6
FIELD PARTY STRING 10
FIELD AMOUNT FLOAT 7
FIELD BALANCE FLOAT 7

TABLE MILL_LEDGER

FIELD ENTRY INTEGER 3
FIELD DATE STRING 6
FIELD DESCRIPTION STRING 20
FIELD TYPE STRING 6
FIELD PARTY STRING 10
FIELD AMOUNT FLOAT 7
```

```
FIELD BALANCE FLOAT 7

TABLE LAST_ENTRIES

FIELD ACCOUNT STRING 7
FIELD ENTRY INTEGER 3
FIELD BALANCE FLOAT 7

DATA "GENERAL" 724 6095.40
DATA "REDWOOD" 281  977.67
DATA "CREEK"   113 1618.26
DATA "MILL"    490 3499.47

END
END
```

```
-) xeq main
SELECT *
FROM   PARCEL_ACCOUNTS

93-293-02  17  821016  DAMAGE FEE  CHARGE  500.00  560.00
93-282-55   1  800101  DUES80      CHARGE  120.00  120.00
93-281-24  31  820107  DUES82      CHARGE  120.00  120.00
93-281-24  32  820107  SA7         CHARGE  240.00  360.00
93-281-24  33  820408  DUES82      CREDIT  120.00  240.00
93-281-24  34  820408  SA7         CREDIT  240.00    0.00
93-281-24  35  820809  SA8         CHARGE  115.00  115.00
93-281-24  36  821105  SA10        CHARGE   72.00  187.00
93-281-24  37  821231  PENALTY82   CHARGE   37.40  224.40
93-282-54  16  820107  DUES82      CHARGE   60.00  370.00
93-282-54  17  820107  SA7         CHARGE  240.00  610.00
93-282-54  18  821231  PENALTY82   CHARGE  122.00  732.00
93-282-55  40  820107  DUES82      CHARGE  120.00  120.00
93-282-55  41  820107  SA7         CHARGE  240.00  360.00
93-282-55  42  820203  DUES82      CREDIT  120.00  240.00
93-282-55  43  820203  SA7         CREDIT  240.00    0.00
92-291-19   7  820107  DUES82      CHARGE   50.00  190.00
92-291-19   8  820107  SA7         CHARGE  240.00  430.00
92-291-19   9  820809  SA8         CHARGE  115.00  545.00
92-291-19  10  821105  SA10        CHARGE   72.00  617.00
92-291-19  11  821123  SA3         CHARGE   42.50  659.50
92-291-19  12  821231  PENALTY82   CHARGE  131.90  791.40
92-291-44  22  820107  DUES82      CHARGE  120.00  120.00
92-291-44  23  821124  DUES82      CREDIT   60.00   60.00
92-291-44  24  821212  DUES82      CREDIT   60.00   60.00
92-293-02   4  820107  DUES82      CHARGE   60.00   60.00
92-293-02   5  820309  DUES82      CREDIT   60.00    0.00
92-293-02   6  821105  SA10        CHARGE   72.00   72.00
92-293-02   7  821119  SA10        CREDIT   72.00    0.00

SELECT *
FROM   PARCEL_ACCOUNTS
WHERE  APN = '93-281-24'

93-281-24  31  820107  DUES82      CHARGE  120.00  120.00
93-281-24  32  820107  SA7         CHARGE  240.00  360.00
93-281-24  33  820408  DUES82      CREDIT  120.00  240.00
93-281-24  34  820408  SA7         CREDIT  240.00    0.00
93-281-24  35  820809  SA8         CHARGE  115.00  115.00
93-281-24  36  821105  SA10        CHARGE   72.00  187.00
93-281-24  37  821231  PENALTY82   CHARGE   37.40  224.40

SELECT *
FROM   PARCEL_ACCOUNTS
WHERE  ENTRY = 7

92-291-19   7  820107  DUES82      CHARGE   50.00  190.00
92-293-02   7  821119  SA10        CREDIT   72.00    0.00

SELECT *
FROM   PARCEL_ACCOUNTS
WHERE  TYPE = 'CHARGE'
AND    AMOUNT = 120.0
```

```
93-282-55   1  800101  DUES80       CHARGE  120.00  120.00
93-281-24  31  820107  DUES82       CHARGE  120.00  120.00
93-282-55  40  820107  DUES82       CHARGE  120.00  120.00
92-291-44  22  820107  DUES82       CHARGE  120.00  120.00

DELETE
FROM    PARCEL_ACCOUNTS
WHERE   TYPE = 'CHARGE'

SELECT *
FROM    PARCEL_ACCOUNTS

93-281-24  33  820408  DUES82       CREDIT  120.00  240.00
93-281-24  34  820408  SA7          CREDIT  240.00    0.00
93-282-55  42  820203  DUES82       CREDIT  120.00  240.00
93-282-55  43  820203  SA7          CREDIT  240.00    0.00
92-291-44  23  821124  DUES82       CREDIT   60.00   60.00
92-291-44  24  821212  DUES82       CREDIT   60.00   60.00
92-293-02   5  820309  DUES82       CREDIT   60.00    0.00
92-293-02   7  821119  SA10         CREDIT   72.00    0.00

DELETE
FROM    PARCEL_ACCOUNTS
WHERE   APN = '93-281-24'
AND     AMOUNT = 120.0

SELECT *
FROM    PARCEL_ACCOUNTS

93-281-24  34  820408  SA7          CREDIT  240.00    0.00
93-282-55  42  820203  DUES82       CREDIT  120.00  240.00
93-282-55  43  820203  SA7          CREDIT  240.00    0.00
92-291-44  23  821124  DUES82       CREDIT   60.00   60.00
92-291-44  24  821212  DUES82       CREDIT   60.00   60.00
92-293-02   5  820309  DUES82       CREDIT   60.00    0.00
92-293-02   7  821119  SA10         CREDIT   72.00    0.00

UPDATE PARCEL_ACCOUNTS
SET     DESCRIPTION = 'BIG BUCKS'
WHERE   AMOUNT = 240.0

SELECT *
FROM    PARCEL_ACCOUNTS

93-281-24  34  820408  BIG BUCKS    CREDIT  240.00    0.00
93-282-55  42  820203  DUES82       CREDIT  120.00  240.00
93-282-55  43  820203  BIG BUCKS    CREDIT  240.00    0.00
92-291-44  23  821124  DUES82       CREDIT   60.00   60.00
92-291-44  24  821212  DUES82       CREDIT   60.00   60.00
92-293-02   5  820309  DUES82       CREDIT   60.00    0.00
92-293-02   7  821119  SA10         CREDIT   72.00    0.00

UPDATE PARCEL_ACCOUNTS
SET     DESCRIPTION = 'DUES82 TOO',
        BALANCE = 0.0
WHERE   APN = '92-291-44'
AND     DATE = '821212'
```

```
SELECT *
FROM   PARCEL_ACCOUNTS

93-281-24  34  820408  BIG BUCKS   CREDIT  240.00    0.00
93-282-55  42  820203  DUES82      CREDIT  120.00  240.00
93-282-55  43  820203  BIG BUCKS   CREDIT  240.00    0.00
92-291-44  23  821124  DUES82      CREDIT   60.00   60.00
92-291-44  24  821212  DUES82 TOO  CREDIT   60.00    0.00
92-293-02   5  820309  DUES82      CREDIT   60.00    0.00
92-293-02   7  821119  SA10        CREDIT   72.00    0.00

UPDATE PARCEL_ACCOUNTS
SET    DESCRIPTION = 'OOPS'

SELECT *
FROM   PARCEL_ACCOUNTS

93-281-24  34  820408  OOPS        CREDIT  240.00    0.00
93-282-55  42  820203  OOPS        CREDIT  120.00  240.00
93-282-55  43  820203  OOPS        CREDIT  240.00    0.00
92-291-44  23  821124  OOPS        CREDIT   60.00   60.00
92-291-44  24  821212  OOPS        CREDIT   60.00    0.00
92-293-02   5  820309  OOPS        CREDIT   60.00    0.00
92-293-02   7  821119  OOPS        CREDIT   72.00    0.00

DELETE
FROM   PARCEL_ACCOUNTS

SELECT *
FROM   PARCEL_ACCOUNTS


INSERT INTO PARCEL_ACCOUNTS ( APN )
       VALUES ('55-555-55')

SELECT *
FROM   PARCEL_ACCOUNTS

55-555-55   0                                 0.00    0.00

INSERT INTO PARCEL_ACCOUNTS ( ENTRY, DATE, APN )
       VALUES (99, '850411', '66-666-66')

SELECT *
FROM   PARCEL_ACCOUNTS

55-555-55   0                                 0.00    0.00
66-666-66  99  850411                         0.00    0.00

SELECT *
FROM   CELLAR

SELECT *
FROM   CELLAR
WHERE  WINE = 'Chardonnay'

SELECT BIN, PRODUCER, READY, BOTTLES
```

```
FROM   CELLAR
WHERE  WINE = 'Chardonnay'

SELECT *
FROM   CELLAR
WHERE  BIN = 3

SELECT CODE
FROM   CITIES
WHERE  CITY = 'San Francisco'

SELECT CODE
FROM   CITIES
WHERE  CITY = 'Chicago'

SELECT *
FROM   FLIGHTS
WHERE  FROM_CODE = 'SFO'
AND    TO_CODE = 'ORD'

SELECT *
FROM   FLIGHTS
WHERE  FROM_CODE =
      (SELECT CODE
       FROM   CITIES
       WHERE  CITY = 'San Francisco')
AND    TO_CODE =
      (SELECT CODE
       FROM   CITIES
       WHERE  CITY = 'Chicago')

SELECT *
FROM   FLIGHTS
WHERE  FROM_CODE =
      (SELECT CODE
       FROM   CITIES
       WHERE  CITY = 'San Francisco')
AND    TO_CODE =
      (SELECT CODE
       FROM   CITIES
       WHERE  CITY = 'Chicago')
ORDER  BY DEP_TIME

SELECT OWNER
FROM   PARCELS
WHERE  APN = '93-282-55'

SELECT AMOUNT
FROM   PARCEL_ACCOUNTS
WHERE  APN = '93-282-55'
AND    DESCRIPTION = 'PENALTY81'
AND    TYPE = 'CHARGE'

SELECT *
FROM   OWNERS
WHERE  ADDRESS LIKE '%BERKELEY%'
```

```
SELECT ENTRY + 1
FROM   LAST_ENTRIES
WHERE  ACCOUNT = 'GENERAL'

SELECT *
FROM   GENERAL_LEDGER
WHERE  PARTY = 'ROAD FIXERS, INC.'
AND    TYPE = 'CHARGE'

SELECT SUM(AMOUNT)
FROM   GENERAL_LEDGER
WHERE  PARTY = 'ROAD FIXERS, INC.'
AND    TYPE = 'CHARGE'

SELECT COUNT(*)
FROM   PARCEL_ACCOUNTS
WHERE  APN = '93-282-55'
AND    TYPE = 'CREDIT'
AND    DATE > '811231'
AND    DATE < '830101'

SELECT MAX(DATE)
FROM   PARCEL_ACCOUNTS
WHERE  APN = '93-282-55'
AND    TYPE = 'CREDIT'

SELECT *
FROM   OWNERS
WHERE  OWNER =
      (SELECT OWNER
       FROM   PARCELS
       WHERE  APN = '93-282-55')

SELECT APN
FROM   PARCELS
WHERE  OWNER = 'JOHN MINSKI'

SELECT SUM(AMOUNT)
FROM   PARCEL_ACCOUNTS
WHERE  TYPE = 'CREDIT'
AND    APN IN ( '93-282-50', '93-282-51', '93-282-54', '93-282-58' )

SELECT SUM(AMOUNT)
FROM   PARCEL_ACCOUNTS
WHERE  TYPE = 'CREDIT'
AND    APN IN
      (SELECT APN
       FROM   PARCELS
       WHERE  OWNER = 'JOHN MINSKI')

SELECT SAN, EXPLANATION, APN
FROM   SPECIAL_ASSESSMENTS, PARCELS
WHERE  SPECIAL_ASSESSMENTS.ROAD = PARCELS.ROAD

SELECT PARCELS.APN, OWNERS.OWNER, OWNERS.ADDRESS, OWNERS.PHONE
FROM   PARCELS, OWNERS
WHERE  PARCELS.IMPROVED = 'Y'
```

```
AND    PARCELS.OWNER = OWNERS.OWNER
ORDER  BY OWNERS.OWNER, PARCELS.APN

SELECT PARCELS.APN, OWNERS.OWNER, OWNERS.ADDRESS, OWNERS.PHONE
FROM   PARCELS, OWNERS
WHERE  PARCELS.IMPROVED = 'Y'
AND    PARCELS.OWNER = OWNERS.OWNER
ORDER  BY OWNERS.OWNER DESC, PARCELS.APN

SELECT APN, OWNER
FROM   PARCELS
WHERE  EXISTS
      (SELECT *
       FROM   PARCEL_ACCOUNTS
       WHERE  APN = PARCELS.APN
       AND    DESCRIPTION = 'DUES82'
       AND    TYPE = 'CREDIT')

SELECT APN, OWNER
FROM   PARCELS
WHERE  NOT EXISTS
      (SELECT *
       FROM   PARCEL_ACCOUNTS
       WHERE  APN = PARCELS.APN
       AND    DESCRIPTION = 'DUES82'
       AND    TYPE = 'CREDIT')

SELECT PARTY, SUM(AMOUNT)
FROM   GENERAL_LEDGER
WHERE  TYPE = 'CHARGE'
GROUP  BY PARTY

SELECT OWNER
FROM   PARCELS
GROUP  BY OWNER
HAVING COUNT(*) > 1

SELECT PARCELS.OWNER, SUM(PARCEL_ACCOUNTS.AMOUNT)
FROM   PARCELS, PARCEL_ACCOUNTS
WHERE  PARCELS.APN = PARCEL_ACCOUNTS.APN
AND    PARCEL_ACCOUNTS.TYPE = 'CREDIT'
AND    PARCEL_ACCOUNTS.DATE LIKE '82%'
GROUP  BY PARCELS.OWNER
HAVING SUM(PARCEL_ACCOUNTS.AMOUNT) > 500
ORDER  BY PARCELS.OWNER

SELECT APN
FROM   PARCELS
WHERE  BALANCE < 0

SELECT OWNER, PHONE
FROM   OWNERS
WHERE  OWNER IN
      (SELECT OWNER
       FROM   PARCELS
       WHERE  BALANCE < 0)
```

```
SELECT AVG(AMOUNT)
FROM   GENERAL_LEDGER
WHERE  DATE LIKE '82%'
AND    TYPE = 'CREDIT'

SELECT PARCELS.APN, PARCELS.ROAD, PARCELS.OWNER, PARCEL_ACCOUNTS.DATE,
       PARCEL_ACCOUNTS.AMOUNT, PARCEL_ACCOUNTS.BALANCE
FROM   PARCELS, PARCEL_ACCOUNTS
WHERE  PARCELS.APN = PARCEL_ACCOUNTS.APN
AND    PARCELS.LAST_ENTRY = PARCEL_ACCOUNTS.ENTRY
ORDER  BY PARCELS.APN

SELECT APN, OWNER
FROM   PARCELS
WHERE  EXISTS
      (SELECT *
       FROM   PARCEL_ACCOUNTS
       WHERE  APN = PARCELS.APN
       AND    TYPE = 'CREDIT'
       AND    AMOUNT > 499.99)

SELECT APN
FROM   PARCEL_ACCOUNTS
WHERE  TYPE = 'CHARGE'
AND    DATE > '801231'
GROUP  BY APN
HAVING COUNT(*) > 5
ORDER  BY APN

-)
```

Appendix III
Modifications to Ada/SQL Binding

As a result of information gathered from a series of Ada/SQL Working Group meetings, changes to the following aspects of the Ada/SQL language have been identified as required for the reasons stated:

1. Naming of generated packages - allow an application scanner to be used to provide required compilation efficiency, without requiring it to change any source code

2. Table definition - avoid the problem of having "generated" table name functions being homographs of the names of the record types defining those tables

3. Exception handling & return status - ensure that the correct status is returned in multi-task applications

4. Explicit type conversion - add the capability, similar to that of Ada, for explicit type conversion between related types. Also remove the implicit type conversion capability envisioned for views.

5. Fixed point arithmetic - expand the definition of Ada/SQL comparable types to allow cross-multiplication and cross-division as with Ada fixed point types. Also provide explicit ways of mapping Ada types into SQL fixed point types.

6. Dynamic SQL - allow data manipulation statements to be built up from individual components at runtime

7. String literals - relax the requirement that column default values be static, so that string literals may be used as defaults

8. Predefined packages - provide stronger definition of support for the predefined packages such as STANDARD, SYSTEM, CALENDAR, etc.

While our in-depth and formal consideration of these areas is continuing, the following informal guidance is provided for current implementations to use for planning purposes:

1. Naming of generated packages - formal proposal contained in Attachment 1

2. Table definition - Change as follows: A <schema package specification> will contain a nested package, ADA_SQL, in which the database related <schema authorization clause>, <schema classification clause>, and <schema declaration element>s must be defined. (<schema classification clause>s may be omitted from early Ada/SQL implementations.) Tables may be declared only within the nested ADA_SQL package; record type declarations placed outside of the nested package do not declare database tables. Each type used for a database column must also be declared within a nested ADA_SQL package. Types may be referenced across schema packages using normal Ada visibility. No restrictions are placed on the Ada declarative items that may be used outside of the nested ADA_SQL package, except that use of names that are the same as database <column name>s, <table identifier>s, or <authorization identifier>s, or other names used by Ada/SQL, may require that those names be expressed as expanded names within Ada/SQL statements if homographs result. Package N_ADA_SQL (see Attachment 1) may be renamed as desired within its corresponding compilation unit, so that the most appropriate prefix for the expanded name may be selected by the programmer.

3. Exception handling & return status - make the effects of the following definitions visible from N_ADA_SQL (the types are effectively defined in a single library package and made visible from each compilation unit's associated N_ADA_SQL package by subtype declarations, so that the same set of types is used by all compilation units within a program):

```
-- The information returned for Ada/SQL errors includes:
-- 1) Context of error, e.g., within what type of statement, and where
--      within that statement, the error occurred (English string)
-- 2) Description of error, in terms of Ada/SQL syntax and/or semantics
--      (e.g., "no table in FROM list contains a column named XXX")
-- 3) Class of error (e.g., SQL statement error, execution of statement
--      would violate uniqueness constraint, etc.)
-- The precise values to be returned for each possible error are to be
--  defined; the formats in which these values are returned are:
```

```
type ERROR_CONTEXT is new STRING;
type ERROR_DESCRIPTION is new STRING;
type ERROR_CLASS is -- enumeration type to be defined

-- The information returned for a single Ada/SQL error is embodied within
--  a single data structure:

type SQLCODE_COMPONENT is private;

-- Since execution of a single Ada/SQL statement can cause several errors,
--  an array is used to return information on all errors caused by
--  executing a statement (a null array is returned if there are no
--  errors):

type SQLCODE_INDEX is new COUNT;
type SQLCODE_ARRAY is array ( SQLCODE_INDEX range <> )
 of SQLCODE_COMPONENT;

-- Since a variable number of errors can be caused by execution of a
--  single Ada/SQL statement, an access type is actually used to return
--  the variable-length array describing those errors:

type SQLCODE_PARAMETER is access SQLCODE_ARRAY;

-- This function returns TRUE if the given SQLCODE_PARAMETER, returned by
--  execution of an Ada/SQL statement, indicates that an error occurred,
--  otherwise FALSE:

function IS_ERROR ( SQLCODE : SQLCODE_PARAMETER ) return BOOLEAN;

-- These functions return the various information items from the data
--  structure for a single Ada/SQL error:

function CONTEXT ( SQLCODE : SQLCODE_COMPONENT ) return ERROR_CONTEXT;
function DESCRIPTION ( SQLCODE : SQLCODE_COMPONENT )
 return ERROR_DESCRIPTION;
function CLASS ( SQLCODE : SQLCODE_COMPONENT ) return ERROR_CLASS;
```

Add a new final out parameter, named SQLCODE and of type SQLCODE_PARAMETER, to all Ada procedures used within the Ada/SQL data manipulation language. This parameter, which is required, is set to indicate the errors that have occurred (if any) in executing the procedure. Exceptions are not raised for error conditions. In another matter related to tasking, require the <cursor name> parameter to the INTO procedures of the <fetch statement>.

Personal note from Fred Friedman: I object to requiring the SQLCODE parameter and the <cursor name> parameters. Instead, I think that they should be optional, so that DML procedures called without a SQLCODE parameter would set a global SQLCODE value, and that <cursor name>s would default to the one used with the FETCH procedure, as originally envisioned in the Ada/SQL specification. Explicit SQLCODE parameters and <cursor name>s are required for the statements to work correctly in a program with tasking; the defaults will work in a program without tasking. I have stated that they are required by the language in accordance with the sentiments of the working group, but here are the arguments pro and con:

The major argument of the working group: The same code should work whether it runs in a program with or without tasking.

The truth: There is no precedent in Ada for this. Any subprogram storing data in uncontrolled persistent variables (a very common programming practice) will have potential synchronization problems when called simultaneously from more than one task. The Ada MIL-STD specifically recognizes the possibility of creating such code (section 9.11), describing its execution as "erroneous", a term which applies to program errors causing unpredictable results, but that cannot be

reasonably prevented by language syntax or checked by a language processor. Furthermore, there is precedent within Ada for defining packages containing two flavors of each subprogram, with and without explicit specification of a data stream indicator. Specifically, several TEXT_IO subprograms may be called with or without a FILE parameter. The FILE parameter may be omitted for convenience in a program without tasking, but would have to be used where several tasks simultaneously input from or output to different files.

My main argument: Making the parameters in question optional allows more streamlined, SQL-like code to be written where tasking is not a consideration, which I suspect will comprise a majority of database applications.

My supporting argument: The precedent is already there in TEXT_IO.

An aside: We might also wish to consider doing something with the <select statement>, which has limitations with respect to tasking as noted in the Ada/SQL specification.

4. Explicit type conversion - define a CONVERT_TO package within N_ADA_SQL (see Attachment 1) as follows (make CONVERT_TO a "reserved word" that cannot be used for a database name): The function to convert database data to type or subtype B declared in schema or predefined package P is visible within N_ADA_SQL as CONVERT_TO.P.B. For example, data in column C may be converted to type B with CONVERT_TO.P.B(C). Strong typing is enforced with explicit type conversions -- conversion is only allowed between entities of compatible classes (e.g., all numeric (sub)types are compatible, all string (sub)types are compatible, two enumeration (sub)types are compatible if and only if one is derived from or a subtype of the other), and any other operations applied to the converted entity are subject to the comparable type checking already defined in the Ada/SQL specification. Note that this form of explicit type conversion may only be applied to database data; the usual Ada type conversion syntax is used, even within Ada/SQL statements, for converting program data. Also, within the <query specification> of a <view definition>, require that the data type of each column be the same as the data type of the corresponding column defined for the view.

5. Fixed point arithmetic - this will be defined at a later date. Early Ada/SQL systems will not be required to support fixed point types. More study of the usefulness of Ada fixed point types within database applications is required.

6. Dynamic SQL - this will be defined at a later date. To simplify initial implementation of application scanners and preprocessors, support for dynamic SQL is not required for the early Ada/SQL systems.

7. String literals - substitute "capable of being computed at compile time" for static

8. Predefined packages - initial Ada/SQL systems should, as a minimum, support the predefined INTEGER, FLOAT, and STRING types. Other requirements will be defined at a later date.

Attachment 1: Proposal with respect to naming of generated packages

1. An "Ada/SQL procedure call" is defined as any of the following Ada/SQL statements:

a. <multiple column unique constraint definition>

b. <view definition>

c. <privilege definition>

d. Any DML statement

2. Let C denote a compilation_unit containing an Ada/SQL procedure call and/or an instantiation of a <correlation name> package.

3. Case:

a. If C defines a library_unit, then:

Case:

1) If the library_unit is a subprogram_declaration or generic_decla ration, then it cannot contain an Ada/SQL procedure call or an instantiation of a <correlation name> package.

2) If the library_unit is a generic_instantiation, then it must be the instantiation of a <correlation name> package. Let N be the <correlation name> (same as the simple_name of the package being declared).

3) If the library_unit is a package_declaration, then:

a) Let S be the simple_name of the package.

b) Let N be the simple_name formed as S_SPEC.

4) If the library_unit is a subprogram_body, then let N be the simple_name of the subprogram.

b. If C defines a secondary_unit then:

Case:

1) If the secondary_unit is a library_unit_body, then let N be the simple_name of the corresponding library_unit.

2) If the secondary_unit is a subunit, then:

a) Let S be the simple_name of the subunit.

b) Let A be the simple_name of its ancestor library unit.

c) Let N be the simple_name formed as A_S.

4. Within a single Ada program library, there shall be at most one compilation_unit producing simple_name N according to the above.

5. A package, N_ADA_SQL, shall be visible from C according to Ada syntax and semantics. (N is as defined in paragraph 3.)

a. A with_clause within the context_clause of C shall name N_ADA_SQL.

b. The name N_ADA_SQL shall not be used in any context clause other than as described herein.

c. The programmer does not write N_ADA_SQL; the package is effectively implemented by the Ada/SQL system.

d. Within an Ada program library, no library_unit shall be written by a user to have a name N_ADA_SQL conflicting with a name produced according to the above.

6. Let D be an Ada/SQL procedure call or instantiation of a <correlation name> package within C. For each such Ada/SQL statement:

a. D shall be within the scope of at least one use_clause denoting N_ADA_SQL.

b. Let T be a database table referenced within D (if any). The reference may be as the <table identifier> within an instantiation of a <correlation name> package, or as a <table name> within an Ada/SQL procedure call. T may be a base table or a viewed table. For each such table referenced:

1) Let P denote the schema package within which table T is declared.

2) C shall not define the specification of P.

3) If C does not define the body of P, then a with_clause naming P shall apply to C.

c. Let O be a <correlation name> referenced (not declared) within D (if any). For each such <correlation name> referenced:

1) Package O shall be directly visible from D.

2) Declarations within O shall not be directly visible from D.

d. Let B be the base type of one of the following:

1) A database column referenced within D (if any).

2) An Ada primary used within D (if any), which is also of the same type as a database column contained within a table referenced by D.

3) A <result program variable> within D (if D is a <fetch statement> or a <select statement>).

e. For each such base type B not defined in the predefined package STANDARD:

1) Let P denote the package in which B is defined.

2) If C does not define the body of P, then a with_clause naming P shall apply to C.

Distribution List for IDA P-1944

NAME AND ADDRESS	NUMBER OF COPIES
Sponsors	
Dr. Michael Mendiville WIS JPMO/ADT Washington, D.C. 20330-6600	5 copies
Other	
Mr. Fred Friedman 7321 Franklin Rd. Annandale, VA 22003-1620	5 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
CSED Review Panel	
Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	1 copy
Dr. Barry W. Boehm TRW Defense Systems Group MS 2-2304 One Space Park Redondo Beach, CA 90278	1 copy
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1 copy
Dr. Larry E. Druffel Software Enginering Institute Shadyside Place 480 South Aiken Av. Pittsburgh, PA 15231	1 copy
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1 copy

Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1 copy
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1 copy
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1 copy

IDA

General W.Y. Smith, HQ	1 copy
Mr. Seymour Deitchman, HQ	1 copy
Ms. Karen H. Weber, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Mr. Bill R. Brykczynski, CSED	50 copies
Ms. Katydean Price, CSED	2 copies
IDA Control & Distribution Vault	3 copies

END

9 — 87

DTIC